Document Number: N4140
Date: 2018-05-03
Revises: N3936
Reply to: Richard Smith
Google Inc
cxxeditor@gmail.com

Working Draft, Standard for Programming Language C++

Note: this is an early draft. It's known to be incomplet and incorrekt, and it has lots of bad fomatting.

Contents

Contents ii

List of Tables iii

List of Figures iv

1 General 1

1.1 Scope . . . 1
1.2 so_8593643_8593643_1 . . . 1
1.3 so_15911714_16006093_3 . . . 1
1.4 Normative references . . . 1
1.5 so_23020323_23021390_8 . . . 1
1.6 so_23020323_23020323_4 . . . 1
1.7 so_23020323_23021390_15 . . . 1
1.8 so_23020323_23020323_2 . . . 1
1.9 so_23020323_23020323_0 . . . 1
1.10 so_13230826_13230826_0 . . . 1
1.11 so_32009189_32009296_0 . . . 1
1.12 so_32009189_32009296_1 . . . 1
1.13 so_37614714_37614932_0 . . . 1
1.14 so_26304506_26304633_2 . . . 1
1.15 so_26304506_26304633_3 . . . 1
1.16 so_26304506_26304633_4 . . . 1
1.17 so_26304506_26304633_0 . . . 1
1.18 so_26304506_26304633_1 . . . 1
1.19 so_38689286_38692249_2 . . . 1
1.20 so_38689286_38692249_3 . . . 1
1.21 so_23802294_23810181_0 . . . 1
1.22 so_23802294_23810181_1 . . . 1
1.23 so_32009189_32009296_2 . . . 1
1.24 so_26304506_26304633_5 . . . 1
1.25 so_41897427_41897885_2 . . . 1
1.26 so_38689286_38692249_4 . . . 1
1.27 Terms and definitions . . . 2
1.28 so_12978745_12978941_0 . . . 2
1.29 so_36670354_36670545_2 . . . 2
1.30 so_22765649_22765649_1 . . . 2
1.31 so_24048701_24088981_2 . . . 2
1.32 so_3892098_3892154_0 . . . 2
1.33 so_17957366_17961043_2 . . . 2
1.34 so_25698780_25698969_0 . . . 2
1.35 so_15805394_15805394_0 . . . 2
1.36 so_15805394_21591062_1 . . . 2
1.37 so_15805394_15805505_4 . . . 2
1.38 so_15805394_15805505_5 . . . 2

1.39 so_11546193_11546378_8 . . . 2
1.40 so_11546193_11546340_2 . . . 2
1.41 so_11546193_11546340_1 . . . 2
1.42 so_11546193_11546340_0 . . . 2
1.43 so_11546193_11546378_1 . . . 2
1.44 so_11546193_11546378_5 . . . 2
1.45 so_11546193_11546378_7 . . . 2
1.46 so_11546193_11546378_9 . . . 2
1.47 so_11546193_11546378_3 . . . 2
1.48 so_11546193_11546378_11 . . . 2
1.49 so_11546193_11546378_4 . . . 2
1.50 so_20694579_20694693_2 . . . 2
1.51 so_36099212_36099281_3 . . . 2
1.52 so_21328655_21328691_1 . . . 2
1.53 so_14189585_14196449_0 . . . 2
1.54 so_13708826_13708826_4 . . . 2
1.55 so_4274763_4274763_0 . . . 2
1.56 so_20712025_20712042_2 . . . 2
1.57 so_16188263_16188602_1 . . . 2
1.58 so_16188263_16188602_2 . . . 2
1.59 so_30057635_30057826_0 . . . 2
1.60 so_20614282_20614325_1 . . . 2
1.61 so_8524989_8525315_0 . . . 2
1.62 so_15258525_15258991_1 . . . 2
1.63 so_46624005_47250175_2 . . . 2
1.64 so_4176328_4176333_3 . . . 2
1.65 so_22950500_22950739_1 . . . 2
1.66 so_22656734_22656791_1 . . . 2
1.67 so_32132574_32132609_0 . . . 2
1.68 so_32132574_32132574_0 . . . 2
1.69 so_27878843_27883301_2 . . . 2
1.70 so_20716091_20716155_3 . . . 2
1.71 so_34418485_34418581_0 . . . 2
1.72 so_33598938_33599142_3 . . . 2
1.73 so_4265167_4272482_0 . . . 2
1.74 so_34559689_34562045_1 . . . 2
1.75 so_34545445_34549139_1 . . . 2
1.76 so_14011707_14011726_0 . . . 2
1.77 so_27988870_27989233_0 . . . 2
1.78 so_24338968_24339068_1 . . . 2
1.79 so_31625914_31626326_0 . . . 2
1.80 so_23542663_23543096_0 . . . 2
1.81 so_10364479_10364487_0 . . . 2
1.82 so_25335467_25335529_0 . . . 2
1.83 so_15959179_15959298_10 . . . 2
1.84 so_33395724_33395724_1 . . . 2
1.85 so_36532687_36532978_0 . . . 2
1.86 so_26290598_26290598_3 . . . 2
1.87 so_32346143_32346399_0 . . . 2
1.88 so_18506029_18506994_0 . . . 2
1.89 so_7468151_7468166_1 . . . 2

1.90 so_30617158_30617677_2 . . . 2
1.91 so_28778672_28778746_2 . . . 2
1.92 so_28778672_28778746_4 . . . 2
1.93 so_24452530_24454652_0 . . . 2
1.94 so_47518542_47518542_4 . . . 2
1.95 so_47518542_47518542_5 . . . 2
1.96 so_20787171_20787187_1 . . . 2
1.97 so_26526426_26526532_0 . . . 2
1.98 so_24215683_24215683_2 . . . 2
1.99 so_340282_341246_3 . . . 2
1.100 so_32761132_32761352_0 . . . 2
1.101 so_3768363_4248640_3 . . . 2
1.102 so_31743001_31743285_1 . . . 2
1.103 so_33074102_33074102_6 . . . 2
1.104 so_17274236_17274384_1 . . . 2
1.105 so_17274236_17274384_2 . . . 2
1.106 so_21110059_21286446_4 . . . 2
1.107 so_21110059_21286446_3 . . . 2
1.108 so_21110059_21286446_5 . . . 2
1.109 so_21110059_21286446_6 . . . 2
1.110 so_6689030_6689041_0 . . . 2
1.111 so_16242400_16242466_0 . . . 2
1.112 so_25999309_25999455_5 . . . 2
1.113 so_24794740_24794786_1 . . . 2
1.114 so_31683281_31683281_0 . . . 2
1.115 so_24032430_24032648_2 . . . 2
1.116 so_24032430_24032648_1 . . . 2
1.117 so_32940847_32945791_4 . . . 2
1.118 so_20829244_20829244_2 . . . 2
1.119 so_20829244_20830550_1 . . . 2
1.120 so_20829244_20830550_0 . . . 2
1.121 so_17290376_17290493_2 . . . 2
1.122 so_27563431_27563466_1 . . . 2
1.123 so_18870156_18870232_0 . . . 2
1.124 so_18870156_18870232_1 . . . 2
1.125 so_15938077_15956145_3 . . . 2
1.126 so_15938077_15956145_1 . . . 2
1.127 so_4351132_4351168_0 . . . 2
1.128 so_4351132_4351168_1 . . . 2
1.129 so_14941723_14941807_0 . . . 2
1.130 so_23119795_23120251_1 . . . 2
1.131 so_5009737_5009814_1 . . . 2
1.132 so_33574112_33574112_1 . . . 2
1.133 so_39548465_39548869_1 . . . 2
1.134 so_14490877_14490951_2 . . . 2
1.135 so_31676842_31677623_1 . . . 2
1.136 so_38342100_38342100_0 . . . 2
1.137 so_1307831_1307866_1 . . . 2
1.138 so_31157995_31158028_0 . . . 2
1.139 so_22941507_22941577_0 . . . 2
1.140 so_14730534_14730687_0 . . . 2

1.141 so_14730534_14731285_2 . . . 2
1.142 so_21133717_21134503_3 . . . 2
1.143 so_27429318_27430786_0 . . . 2
1.144 so_20873932_20873970_0 . . . 2
1.145 so_10792757_10792766_1 . . . 2
1.146 so_15231527_15231718_0 . . . 2
1.147 so_36021322_36021621_0 . . . 2
1.148 so_36021322_36021642_1 . . . 2
1.149 so_7649649_7649649_1 . . . 2
1.150 so_7649649_7649649_0 . . . 2
1.151 so_4042626_4042686_1 . . . 2
1.152 so_14420908_14420908_10 . . . 2
1.153 so_14420908_14420908_7 . . . 2
1.154 so_14420908_14420908_6 . . . 2
1.155 so_6171814_6171814_1 . . . 2
1.156 so_25300461_25300537_0 . . . 2
1.157 so_10528952_10529036_0 . . . 2
1.158 so_290038_290048_2 . . . 2
1.159 so_290038_290048_1 . . . 2
1.160 so_16428234_16428253_0 . . . 2
1.161 so_16428234_16428253_2 . . . 2
1.162 so_2046952_2047172_1 . . . 2
1.163 so_2046952_2047172_5 . . . 2
1.164 so_2046952_2047172_3 . . . 2
1.165 so_8599225_8599332_3 . . . 2
1.166 so_46894136_46894648_1 . . . 2
1.167 so_19033448_19037004_1 . . . 2
1.168 so_47244441_47244441_5 . . . 2
1.169 so_47244441_47244441_2 . . . 2
1.170 so_32589328_32589328_2 . . . 2
1.171 so_32589328_32589328_3 . . . 2
1.172 so_11240484_11369012_0 . . . 2
1.173 so_24186681_32434618_5 . . . 2
1.174 so_28466069_28466236_3 . . . 2
1.175 so_6716091_6716109_0 . . . 2
1.176 so_33984217_33984222_0 . . . 2
1.177 so_20355948_20356073_3 . . . 2
1.178 so_10492513_10492522_0 . . . 2
1.179 so_24296571_24296629_1 . . . 2
1.180 so_24296571_24297811_2 . . . 2
1.181 so_21139019_21139234_1 . . . 2
1.182 so_1468058_1468304_0 . . . 2
1.183 so_19636539_19636588_1 . . . 2
1.184 so_2397984_2397995_3 . . . 2
1.185 so_2397984_23581132_0 . . . 2
1.186 so_2397984_23581132_1 . . . 2
1.187 so_2397984_4105123_5 . . . 2
1.188 so_2397984_23581132_2 . . . 2
1.189 so_23153445_23154502_2 . . . 2
1.190 so_9212212_9212392_1 . . . 2
1.191 so_39720871_39721566_0 . . . 2

1.192 so_29821811_29822863_4 . . . 2
1.193 so_32154832_32157764_0 . . . 2
1.194 so_33916113_33916180_1 . . . 2
1.195 so_33916113_33916186_3 . . . 2
1.196 Implementation compliance . . . 5
1.197 so_17957366_17961043_0 . . . 5
1.198 so_15805394_15805532_6 . . . 5
1.199 so_18727022_18727216_3 . . . 5
1.200 so_5177167_5177289_1 . . . 5
1.201 so_15935073_15936695_1 . . . 5
1.202 so_2301372_2301426_0 . . . 5
1.203 so_2128321_2128727_2 . . . 5
1.204 so_26309910_26310017_1 . . . 5
1.205 so_26237432_26237876_1 . . . 5
1.206 so_41869849_41870007_2 . . . 5
1.207 so_6589946_6592141_0 . . . 5
1.208 so_15805394_15805532_5 . . . 5
1.209 so_15805394_15805505_2 . . . 5
1.210 so_15805394_15805505_0 . . . 5
1.211 so_15805394_21591062_0 . . . 5
1.212 so_15805394_15805532_4 . . . 5
1.213 so_15805394_15805505_1 . . . 5
1.214 so_15805394_15805532_3 . . . 5
1.215 so_27629816_27630346_4 . . . 5
1.216 so_31685132_31685448_9 . . . 5
1.217 so_30438718_30438718_0 . . . 5
1.218 so_22180312_22180312_1 . . . 5
1.219 so_22180312_22180312_0 . . . 5
1.220 so_34615352_34615529_0 . . . 5
1.221 so_31681856_31682099_3 . . . 5
1.222 so_31681856_31682099_1 . . . 5
1.223 so_31681856_31682099_0 . . . 5
1.224 so_31681856_31682099_2 . . . 5
1.225 so_31681856_31682129_0 . . . 5
1.226 so_5177167_5177289_2 . . . 5
1.227 so_5177167_5177289_3 . . . 5
1.228 so_5177167_5177289_4 . . . 5
1.229 so_15935073_15936695_3 . . . 5
1.230 so_45497684_45522989_1 . . . 5
1.231 so_45497684_45522989_0 . . . 5
1.232 so_46011046_46011046_0 . . . 5
1.233 so_2532912_39442655_1 . . . 5
1.234 so_25300461_25300537_1 . . . 5
1.235 so_2128321_2128727_3 . . . 5
1.236 so_28466069_28466236_1 . . . 5
1.237 so_28466069_28466236_0 . . . 5
1.238 so_38486235_38486324_1 . . . 5
1.239 so_41982545_41983799_1 . . . 5
1.240 so_16876410_21311124_3 . . . 5
1.241 so_18551174_18552104_1 . . . 5
1.242 so_26811356_26811447_2 . . . 5

1.243 so_13902910_13912596_1 . . . 5
1.244 so_12750531_12750695_0 . . . 5
1.245 so_12750531_12750695_1 . . . 5
1.246 so_28085557_28086078_0 . . . 5
1.247 so_27777071_27807579_2 . . . 5
1.248 so_2776381_2776639_0 . . . 5
1.249 so_204476_18721336_27 . . . 5
1.250 so_204476_31263079_4 . . . 5
1.251 so_204476_31263079_12 . . . 5
1.252 so_8956160_8956177_0 . . . 5
1.253 so_42441683_42442198_1 . . . 5
1.254 so_7050925_7050940_1 . . . 5
1.255 so_27419071_27419648_1 . . . 5
1.256 so_15805394_15805474_0 . . . 5
1.257 so_27629816_27630346_5 . . . 5
1.258 so_8578791_8579314_2 . . . 5
1.259 so_21963734_21963998_4 . . . 5
1.260 so_34615352_34615529_1 . . . 5
1.261 so_41723401_41724629_1 . . . 5
1.262 so_204476_18721336_28 . . . 5
1.263 so_22182432_22182432_0 . . . 5
1.264 so_46011046_46011067_0 . . . 5
1.265 so_32270244_32270723_2 . . . 5
1.266 so_21265462_21267714_1 . . . 5
1.267 so_23553723_23553744_0 . . . 5
1.268 so_28466069_28466464_0 . . . 5
1.269 so_28114301_28117559_3 . . . 5
1.270 so_35730650_35730741_3 . . . 6
1.271 so_204476_18721336_29 . . . 6
1.272 so_204476_18721336_30 . . . 6
1.273 Structure of this International Standard . . . 6
1.274 Syntax notation . . . 6
1.275 so_13754778_13758060_0 . . . 6
1.276 so_13754778_13754778_0 . . . 6
1.277 The C++ memory model . . . 7
1.278 so_41189577_41190370_0 . . . 7
1.279 so_14256695_14257140_0 . . . 7
1.280 so_3056174_3056631_2 . . . 7
1.281 so_35852493_35853481_4 . . . 7
1.282 so_35852493_35853459_0 . . . 7
1.283 so_33570247_33570352_1 . . . 7
1.284 so_15604411_15604573_0 . . . 7
1.285 so_4266771_4266820_1 . . . 7
1.286 so_4329777_4329777_2 . . . 7
1.287 so_12655875_12655875_0 . . . 7
1.288 so_13819820_13819885_0 . . . 7
1.289 so_34512747_35002931_1 . . . 7
1.290 so_29338126_29339251_1 . . . 7
1.291 so_29338126_29338173_0 . . . 7
1.292 so_876213_876216_0 . . . 7
1.293 so_42632224_42632606_0 . . . 7

1.294 so_22430998_22431501_3 . . . 7
1.295 so_15934111_15943751_1 . . . 7
1.296 so_34737737_34738084_2 . . . 7
1.297 so_34737737_34738084_1 . . . 7
1.298 so_30395840_30396103_0 . . . 7
1.299 so_22824974_22825016_0 . . . 7
1.300 so_34122434_34122776_4 . . . 7
1.301 so_34706564_34706615_1 . . . 7
1.302 so_34706564_34706615_0 . . . 7
1.303 so_47578538_47579547_0 . . . 7
1.304 so_19761707_19764768_0 . . . 7
1.305 so_25697057_25697207_0 . . . 7
1.306 so_20548528_20548675_0 . . . 7
1.307 so_47193195_47193591_0 . . . 7
1.308 so_47193195_47193591_2 . . . 7
1.309 so_18303212_18303363_0 . . . 7
1.310 so_10897093_10897093_0 . . . 7
1.311 so_27705409_27734485_0 . . . 7
1.312 so_13484113_13484201_1 . . . 7
1.313 so_24152858_24155873_0 . . . 7
1.314 so_39256401_39256595_0 . . . 7
1.315 so_3674929_3674978_0 . . . 7
1.316 so_18303212_18303363_2 . . . 7
1.317 so_30337439_30337439_2 . . . 7
1.318 so_30337439_30337439_3 . . . 7
1.319 The C++ object model . . . 7
1.320 so_43275410_43275782_0 . . . 7
1.321 so_43275410_43275782_2 . . . 7
1.322 so_43275410_43275782_1 . . . 7
1.323 so_15945298_15945444_1 . . . 7
1.324 so_35852493_35853481_1 . . . 7
1.325 so_32700032_32700032_1 . . . 7
1.326 so_28795540_28795540_0 . . . 7
1.327 so_25674834_25675061_0 . . . 7
1.328 so_36024325_36024383_0 . . . 7
1.329 so_1745487_1745667_1 . . . 7
1.330 so_15604411_15604573_1 . . . 7
1.331 so_26533740_26535126_4 . . . 7
1.332 so_43971298_43971298_0 . . . 7
1.333 so_16251127_16251288_0 . . . 7
1.334 so_46909105_46910105_0 . . . 7
1.335 so_43946981_43947032_1 . . . 7
1.336 so_43946981_43947251_2 . . . 7
1.337 so_43946981_43947251_0 . . . 7
1.338 so_43946981_43947251_1 . . . 7
1.339 so_15938077_15938077_1 . . . 7
1.340 so_14811171_14811849_0 . . . 7
1.341 so_35897263_35897408_0 . . . 7
1.342 so_42246545_42246733_0 . . . 7
1.343 so_5129312_5129337_0 . . . 7
1.344 so_5129312_5129407_0 . . . 7

1.345 so_24577997_24578063_1 . . . 7
1.346 so_30114397_30114921_4 . . . 7
1.347 so_20850536_20999389_2 . . . 7
1.348 so_46960774_46961291_0 . . . 7
1.349 so_20105329_20105367_0 . . . 7
1.350 so_46838729_46838940_3 . . . 7
1.351 so_9929703_9929734_0 . . . 7
1.352 so_38267148_39675770_0 . . . 8
1.353 so_19132671_19132953_0 . . . 8
1.354 so_40108184_40108184_0 . . . 8
1.355 so_38226443_38227096_1 . . . 8
1.356 so_38226443_38226983_2 . . . 8
1.357 so_25562042_25562267_0 . . . 8
1.358 so_25562042_25562868_3 . . . 8
1.359 so_46738487_46743940_3 . . . 8
1.360 so_44909407_44909130_2 . . . 8
1.361 so_26781317_26782633_0 . . . 8
1.362 so_33574112_33583262_1 . . . 8
1.363 so_23498746_23499592_0 . . . 8
1.364 so_18451683_18452260_0 . . . 8
1.365 so_25562042_25562868_4 . . . 8
1.366 so_25562042_25562868_6 . . . 8
1.367 so_25562042_25562267_1 . . . 8
1.368 so_25562042_25562868_5 . . . 8
1.369 so_33574112_33583262_3 . . . 8
1.370 so_33574112_33583262_2 . . . 8
1.371 so_33574112_33583262_4 . . . 8
1.372 so_14811171_14811849_1 . . . 8
1.373 so_33574112_33583262_5 . . . 8
1.374 so_12241637_12241637_0 . . . 8
1.375 so_18451683_18452260_1 . . . 8
1.376 so_29777492_39021760_8 . . . 8
1.377 so_29777492_29866102_0 . . . 8
1.378 so_621616_621638_0 . . . 8
1.379 so_2826589_2826589_1 . . . 8
1.380 so_2826589_2826863_0 . . . 8
1.381 so_7269369_7269793_0 . . . 8
1.382 so_19949737_19950151_1 . . . 8
1.383 so_12138243_12181316_0 . . . 8
1.384 so_12138243_12175516_2 . . . 8
1.385 so_44909407_44909130_4 . . . 8
1.386 so_4351132_4351132_1 . . . 8
1.387 so_14811171_14811849_2 . . . 8
1.388 so_3902513_3902513_1 . . . 8
1.389 so_44187821_44188162_1 . . . 8
1.390 so_42632224_42632606_1 . . . 8
1.391 so_14812214_14812252_0 . . . 8
1.392 so_42608320_42608531_1 . . . 8
1.393 so_26227965_26228269_0 . . . 8
1.394 so_44914678_44914988_1 . . . 8
1.395 so_14092284_14092302_1 . . . 8

1.396 so_46342838_46347559_0 . . . 8
1.397 so_46745805_46747412_0 . . . 8
1.398 so_4657976_4658026_0 . . . 8
1.399 so_38082969_38083379_0 . . . 8
1.400 so_7760198_7760516_0 . . . 8
1.401 so_7760198_7760198_0 . . . 8
1.402 so_35852493_35853481_2 . . . 8
1.403 so_35852493_35853459_1 . . . 8
1.404 so_13484113_13484201_0 . . . 8
1.405 so_19949737_19949872_0 . . . 8
1.406 so_11300439_11300596_0 . . . 8
1.407 so_26533740_26535126_2 . . . 8
1.408 so_26533740_26535126_3 . . . 8
1.409 so_32085371_32085434_0 . . . 8
1.410 so_17341948_17342262_0 . . . 8
1.411 so_46735324_46736040_2 . . . 8
1.412 so_24252464_24252692_2 . . . 8
1.413 so_24252464_24252692_3 . . . 8
1.414 so_44909407_44909913_0 . . . 8
1.415 so_44909407_44909130_6 . . . 8
1.416 so_44909407_44909831_0 . . . 8
1.417 so_3902513_3902513_3 . . . 8
1.418 so_3902513_3902513_2 . . . 8
1.419 so_13506734_13507227_0 . . . 8
1.420 so_44187821_44188162_2 . . . 8
1.421 so_40873520_40874245_1 . . . 8
1.422 so_46600017_46600272_0 . . . 8
1.423 so_5246969_5247009_4 . . . 8
1.424 so_5246969_5247799_0 . . . 8
1.425 so_23498746_23502492_2 . . . 8
1.426 so_23498746_23502492_0 . . . 8
1.427 so_44914678_44914988_2 . . . 8
1.428 so_14092284_14092302_2 . . . 8
1.429 so_14092284_14092302_3 . . . 8
1.430 Program execution . . . 8
1.431 so_8822229_8918505_0 . . . 8
1.432 so_26751895_26752006_4 . . . 8
1.433 so_2535148_2535246_1 . . . 8
1.434 so_47531139_47531235_0 . . . 8
1.435 so_15718262_15718279_1 . . . 8
1.436 so_15718262_15718279_0 . . . 8
1.437 so_41857372_41957932_2 . . . 8
1.438 so_16233497_16235467_0 . . . 8
1.439 so_37083614_37085442_3 . . . 8
1.440 so_42179740_42213312_0 . . . 8
1.441 so_42540954_42541005_2 . . . 8
1.442 so_6893116_6893229_1 . . . 8
1.443 so_32413700_32416123_1 . . . 8
1.444 so_26190364_26194081_1 . . . 8
1.445 so_26190364_26190816_0 . . . 8
1.446 so_2306587_2307021_0 . . . 8

1.447 so_2306587_2307021_1 . . . 8
1.448 so_9957695_9957704_1 . . . 8
1.449 so_9957695_9957704_2 . . . 8
1.450 so_15863421_15863504_0 . . . 8
1.451 so_14785639_14785639_1 . . . 8
1.452 so_25847349_25847445_5 . . . 8
1.453 so_25847349_25847445_4 . . . 8
1.454 so_39340168_39342355_3 . . . 8
1.455 so_39340168_39342355_2 . . . 8
1.456 so_39914788_39925056_1 . . . 8
1.457 so_26771692_26782247_1 . . . 8
1.458 so_26771692_26782247_0 . . . 8
1.459 so_6364365_6364568_0 . . . 8
1.460 so_15777133_15779613_2 . . . 8
1.461 so_15777133_15779613_1 . . . 8
1.462 so_33127973_33128231_1 . . . 8
1.463 so_10521856_10522106_5 . . . 8
1.464 so_10521856_10522106_4 . . . 8
1.465 so_4796270_4796323_1 . . . 8
1.466 so_16732788_16732864_1 . . . 8
1.467 so_31873616_31875379_0 . . . 8
1.468 so_31873616_31877074_1 . . . 8
1.469 so_31873616_31877074_2 . . . 8
1.470 so_32845759_32848265_1 . . . 8
1.471 so_32845759_32848265_0 . . . 8
1.472 so_5563000_8935697_1 . . . 8
1.473 so_24210174_24210487_0 . . . 8
1.474 so_26013183_26013641_0 . . . 8
1.475 so_36066000_36066446_1 . . . 8
1.476 so_35648348_35654696_0 . . . 8
1.477 so_35648348_35654696_1 . . . 8
1.478 so_3372703_3375403_3 . . . 8
1.479 so_45347308_45349486_1 . . . 8
1.480 so_26918903_26919328_5 . . . 8
1.481 so_24186681_32434618_12 . . . 8
1.482 so_24186681_32434618_7 . . . 8
1.483 so_2397984_4105123_1 . . . 8
1.484 so_33260200_33284053_1 . . . 8
1.485 so_33260200_33284053_0 . . . 8
1.486 so_32154832_32154832_1 . . . 8
1.487 so_29381223_29381539_0 . . . 8
1.488 so_35730650_35730741_5 . . . 9
1.489 so_32132574_32141702_0 . . . 9
1.490 so_24186681_32434618_13 . . . 9
1.491 so_24186681_32434618_8 . . . 9
1.492 so_2397984_4105123_2 . . . 9
1.493 so_9116036_9116162_0 . . . 9
1.494 so_22248587_22248587_0 . . . 9
1.495 so_621542_621548_0 . . . 9
1.496 so_24186681_32434618_9 . . . 9
1.497 so_18516510_18516712_0 . . . 9

1.498 so_2397984_4105123_3 . . . 9
1.499 so_15718262_15718279_3 . . . 9
1.500 so_4364536_4364586_1 . . . 9
1.501 so_9738471_9738507_0 . . . 9
1.502 so_24794740_24794740_1 . . . 9
1.503 so_3932081_3934394_0 . . . 9
1.504 so_24186681_32434618_10 . . . 9
1.505 so_2397984_4105123_4 . . . 9
1.506 so_23153445_23153645_0 . . . 9
1.507 so_23153445_23153722_0 . . . 9
1.508 so_14154382_14154812_0 . . . 9
1.509 so_2727834_2727883_0 . . . 9
1.510 so_29777492_29777728_1 . . . 9
1.511 so_8822229_8918505_1 . . . 9
1.512 so_8287010_8287110_1 . . . 9
1.513 so_15718262_15718279_2 . . . 9
1.514 so_41857372_41957932_1 . . . 9
1.515 so_25284863_25285719_2 . . . 9
1.516 so_25287484_25287819_2 . . . 9
1.517 so_39914788_39940251_0 . . . 9
1.518 so_39914788_39915175_0 . . . 9
1.519 so_6364365_6364568_1 . . . 9
1.520 so_33127973_33128231_2 . . . 9
1.521 so_21110059_21286446_2 . . . 9
1.522 so_34907922_34907922_1 . . . 9
1.523 so_4796270_4796323_0 . . . 9
1.524 so_29326097_29326353_0 . . . 9
1.525 so_36814199_36814227_0 . . . 9
1.526 so_32589328_32589328_4 . . . 9
1.527 so_24186681_24187757_1 . . . 9
1.528 so_24186681_32434618_11 . . . 9
1.529 so_30437973_30438534_0 . . . 9
1.530 so_23153445_23154502_1 . . . 9
1.531 so_23153445_23153645_1 . . . 9
1.532 so_36613587_36613757_3 . . . 9
1.533 so_29708242_29708410_2 . . . 9
1.534 so_4136900_4137115_4 . . . 9
1.535 so_21415700_21417377_0 . . . 9
1.536 so_1745487_1745667_0 . . . 9
1.537 so_2196327_2196495_1 . . . 9
1.538 so_21592083_21592083_0 . . . 9
1.539 so_8822229_8918505_2 . . . 9
1.540 so_26751895_26752006_7 . . . 9
1.541 so_26751895_26752006_5 . . . 9
1.542 so_34863148_34863709_2 . . . 9
1.543 so_34863148_34863709_3 . . . 9
1.544 so_34863148_34863709_0 . . . 9
1.545 so_34863148_34863709_1 . . . 9
1.546 so_2535148_2535232_2 . . . 9
1.547 so_29268092_29268092_1 . . . 9
1.548 so_29268092_29268092_3 . . . 9

1.549 so_29268092_29268092_0 . . . 9
1.550 so_29268092_29268092_2 . . . 9
1.551 so_15718262_15718279_7 . . . 9
1.552 so_15718262_15718279_6 . . . 9
1.553 so_15718262_46455917_0 . . . 9
1.554 so_15718262_15718279_5 . . . 9
1.555 so_15718262_15718279_4 . . . 9
1.556 so_15718262_15718279_8 . . . 9
1.557 so_1490262_1490282_1 . . . 9
1.558 so_16233497_16235467_2 . . . 9
1.559 so_16233497_16235467_1 . . . 9
1.560 so_16233497_16235467_3 . . . 9
1.561 so_4136900_4137115_7 . . . 9
1.562 so_4136900_4137115_8 . . . 9
1.563 so_4136900_4137115_5 . . . 9
1.564 so_20080362_20080517_0 . . . 9
1.565 so_46994763_46994830_0 . . . 9
1.566 so_6130100_6130497_0 . . . 9
1.567 so_37083614_37085442_5 . . . 9
1.568 so_26190364_26194081_3 . . . 9
1.569 so_26190364_26194081_2 . . . 9
1.570 so_15863421_15863504_3 . . . 9
1.571 so_15863421_15863504_1 . . . 9
1.572 so_15863421_15863504_2 . . . 9
1.573 so_32900370_32900419_2 . . . 9
1.574 so_14785639_14785639_2 . . . 9
1.575 so_25298157_25298157_1 . . . 9
1.576 so_25298157_25298191_0 . . . 9
1.577 so_8952593_8952728_2 . . . 9
1.578 so_21110059_21250206_0 . . . 9
1.579 so_21110059_21250206_1 . . . 9
1.580 so_21110059_21273622_0 . . . 9
1.581 so_21110059_21250206_2 . . . 9
1.582 so_21110059_21273622_1 . . . 9
1.583 so_21110059_21250206_3 . . . 9
1.584 so_3785798_3785944_0 . . . 9
1.585 so_3785798_3785944_1 . . . 9
1.586 so_3785798_3789617_2 . . . 9
1.587 so_3785798_3789617_0 . . . 9
1.588 so_5487925_5488316_0 . . . 9
1.589 so_5487925_5488316_1 . . . 9
1.590 so_16732788_16732864_2 . . . 9
1.591 so_16732788_16732864_3 . . . 9
1.592 so_16732788_16732864_4 . . . 9
1.593 so_16732788_16732864_5 . . . 9
1.594 so_26230791_26234699_1 . . . 9
1.595 so_36814199_36814227_2 . . . 9
1.596 so_36814199_36814227_1 . . . 9
1.597 so_15880917_15882787_1 . . . 9
1.598 so_15880917_15882787_0 . . . 9
1.599 so_598148_599564_6 . . . 9

1.600 so_598148_599564_5 . . . 9
1.601 so_39824814_39824879_3 . . . 10
1.602 so_38954300_38954300_4 . . . 10
1.603 so_45744673_45744673_2 . . . 10
1.604 so_45744673_45744673_3 . . . 10
1.605 so_16291127_16292092_1 . . . 10
1.606 so_39340168_39340220_0 . . . 10
1.607 so_34131331_34131610_0 . . . 10
1.608 so_19881803_19881804_0 . . . 10
1.609 so_23557419_23557641_1 . . . 10
1.610 so_31206205_31207433_1 . . . 10
1.611 so_33828350_33829487_4 . . . 10
1.612 so_39025342_39025722_0 . . . 10
1.613 so_45576309_45576309_5 . . . 10
1.614 so_45576309_45576309_6 . . . 10
1.615 so_45576309_45576309_4 . . . 10
1.616 so_28280169_28280284_1 . . . 11
1.617 so_12554619_12554621_2 . . . 11
1.618 so_37919291_37919534_3 . . . 11
1.619 so_8822229_8918505_3 . . . 11
1.620 so_29268092_29268092_5 . . . 11
1.621 so_26805502_26805636_1 . . . 11
1.622 so_1978709_24137427_1 . . . 11
1.623 so_34840115_34840115_0 . . . 11
1.624 so_12881285_12881285_0 . . . 11
1.625 so_12881285_12881285_1 . . . 11
1.626 so_3852768_3852768_0 . . . 11
1.627 so_3852768_3852774_2 . . . 11
1.628 so_46994763_46994830_2 . . . 11
1.629 so_14785639_14785760_0 . . . 11
1.630 so_35729956_35730320_0 . . . 11
1.631 so_4369219_4369219_0 . . . 11
1.632 so_45753903_45753903_0 . . . 11
1.633 so_21110059_21286446_1 . . . 11
1.634 so_3785798_3785944_3 . . . 11
1.635 so_9563600_9563669_0 . . . 11
1.636 so_17400137_17403467_0 . . . 11
1.637 so_32940847_32941074_2 . . . 11
1.638 so_32940847_32945791_2 . . . 11
1.639 so_24194076_24206844_1 . . . 11
1.640 so_24194076_24218798_3 . . . 11
1.641 so_36816894_36817210_1 . . . 11
1.642 so_46960774_46961420_4 . . . 11
1.643 so_708742_708945_1 . . . 11
1.644 so_36099212_36099281_1 . . . 11
1.645 so_27835808_27836620_1 . . . 11
1.646 so_25287484_25287819_0 . . . 11
1.647 so_25287484_25287484_0 . . . 11
1.648 so_17443163_17443575_1 . . . 11
1.649 so_15777133_15779613_3 . . . 11
1.650 so_21670459_46186122_0 . . . 11

1.651 so_10778627_10778627_0 . . . 11
1.652 so_30691551_30691660_1 . . . 11
1.653 so_38215743_38215816_1 . . . 11
1.654 so_26918903_26919328_3 . . . 11
1.655 so_39340030_39340200_1 . . . 11
1.656 so_29513572_29513843_1 . . . 11
1.657 so_29513572_29513843_2 . . . 11
1.658 so_19231779_19232158_1 . . . 11
1.659 so_24186681_24186681_0 . . . 11
1.660 so_5214611_5214681_0 . . . 11
1.661 so_4176328_4183735_0 . . . 11
1.662 so_26190364_26194081_0 . . . 11
1.663 so_22076038_22076139_0 . . . 11
1.664 so_24224115_24226345_5 . . . 11
1.665 so_39340168_39340220_1 . . . 11
1.666 so_15777133_15779613_4 . . . 11
1.667 so_38043932_38043989_3 . . . 11
1.668 so_6687390_6687496_0 . . . 11
1.669 so_10778627_10778627_1 . . . 11
1.670 so_30267421_30267466_0 . . . 11
1.671 so_31206205_31207433_0 . . . 11
1.672 so_33828350_33829487_6 . . . 11
1.673 so_44988879_44988959_0 . . . 11
1.674 so_13744507_13745174_0 . . . 11
1.675 so_9777561_9777705_0 . . . 11
1.676 so_47305013_47305053_1 . . . 11
1.677 so_22329703_22330014_1 . . . 11
1.678 so_37786547_37786733_0 . . . 11
1.679 so_43479839_43480231_3 . . . 11
1.680 so_26707897_26708137_0 . . . 11
1.681 so_21554577_21554578_3 . . . 11
1.682 so_9964786_9964844_0 . . . 11
1.683 so_35464793_35464872_1 . . . 11
1.684 so_33131966_33132074_0 . . . 11
1.685 so_3850040_3850092_1 . . . 11
1.686 so_5523244_5523372_2 . . . 11
1.687 so_19729591_19729681_3 . . . 11
1.688 so_36099212_36099281_2 . . . 11
1.689 so_31560073_31562511_1 . . . 11
1.690 so_31560073_31562511_2 . . . 11
1.691 so_27835808_27839451_0 . . . 11
1.692 so_27835808_27848818_3 . . . 11
1.693 so_27835808_27839451_2 . . . 11
1.694 so_27835808_27836620_0 . . . 11
1.695 so_27835808_27836016_0 . . . 11
1.696 so_27835808_27848818_1 . . . 11
1.697 so_27835808_27836016_1 . . . 11
1.698 so_26694842_26694904_0 . . . 11
1.699 so_26694842_26694904_2 . . . 11
1.700 so_26694842_26694904_3 . . . 11
1.701 so_32866759_32866905_0 . . . 11

1.702 so_36515046_36515302_0 . . . 11
1.703 so_33291146_33291213_1 . . . 11
1.704 so_3852768_3852774_1 . . . 11
1.705 so_3852768_3852768_1 . . . 11
1.706 so_3852768_3852774_5 . . . 11
1.707 so_3852768_3852768_2 . . . 11
1.708 so_44455682_44491564_1 . . . 11
1.709 so_4136900_4137115_2 . . . 11
1.710 so_35120763_35204414_0 . . . 11
1.711 so_4176328_4183735_1 . . . 11
1.712 so_4176328_4183735_2 . . . 11
1.713 so_4176328_4183735_6 . . . 11
1.714 so_4176328_4183735_4 . . . 11
1.715 so_4176328_4183735_8 . . . 11
1.716 so_4176328_4183735_3 . . . 11
1.717 so_996844_997206_0 . . . 11
1.718 so_30539849_30539954_1 . . . 11
1.719 so_29316538_29316636_0 . . . 11
1.720 so_29316538_29316641_1 . . . 11
1.721 so_29316538_29316641_2 . . . 11
1.722 so_29316538_29316641_0 . . . 11
1.723 so_25466285_25466478_0 . . . 11
1.724 so_25466285_25466314_0 . . . 11
1.725 so_42540954_42541028_0 . . . 11
1.726 so_42540954_42541000_0 . . . 11
1.727 so_34168897_34168959_0 . . . 11
1.728 so_34168897_34168982_0 . . . 11
1.729 so_28155039_28156035_0 . . . 11
1.730 so_30694923_30694995_0 . . . 11
1.731 so_9566187_9566538_1 . . . 11
1.732 so_33598938_33600274_4 . . . 11
1.733 so_33598938_33600274_2 . . . 11
1.734 so_46071978_46072243_0 . . . 11
1.735 so_24326572_24327094_4 . . . 11
1.736 so_40827303_40827376_3 . . . 11
1.737 so_40827303_40827376_2 . . . 11
1.738 so_40827303_40827376_1 . . . 11
1.739 so_27470421_27470487_0 . . . 11
1.740 so_7914972_7915003_1 . . . 11
1.741 so_2106834_2107128_0 . . . 11
1.742 so_22642699_22643080_0 . . . 11
1.743 so_36133311_36133565_1 . . . 11
1.744 so_38837592_38837675_0 . . . 11
1.745 so_46331625_46331733_0 . . . 11
1.746 so_31063331_31063331_3 . . . 11
1.747 so_31063331_31063331_1 . . . 11
1.748 so_31063331_31063723_0 . . . 11
1.749 so_31063331_31063331_0 . . . 11
1.750 so_32504524_32514058_2 . . . 11
1.751 so_32504524_32514058_3 . . . 11
1.752 so_32504524_32514058_0 . . . 11

1.753 so_32504524_32504762_0 . . . 11
1.754 so_32504524_32514058_1 . . . 11
1.755 so_32504524_32504764_0 . . . 11
1.756 so_15166907_15168020_2 . . . 11
1.757 so_19069681_19069981_0 . . . 11
1.758 so_3927936_3928128_0 . . . 11
1.759 so_39914342_39914727_1 . . . 11
1.760 so_11060968_11060994_0 . . . 11
1.761 so_23703601_23703753_0 . . . 11
1.762 so_10655290_10655884_4 . . . 11
1.763 so_10655290_10655884_3 . . . 11
1.764 so_10655290_10655884_0 . . . 11
1.765 so_5326491_5326547_1 . . . 11
1.766 so_44491112_44491199_0 . . . 11
1.767 so_20800684_20800786_0 . . . 11
1.768 so_20800684_20800762_0 . . . 11
1.769 so_20800684_20805797_7 . . . 11
1.770 so_20800684_20805797_9 . . . 11
1.771 so_35042548_35043929_0 . . . 11
1.772 so_38043932_38043989_4 . . . 11
1.773 so_6687390_6687496_1 . . . 11
1.774 so_6687390_6687496_2 . . . 11
1.775 so_21670459_21670459_0 . . . 11
1.776 so_21670459_21670570_0 . . . 11
1.777 so_21670459_21670459_3 . . . 11
1.778 so_21670459_46186122_1 . . . 11
1.779 so_21670459_21670751_0 . . . 11
1.780 so_21670459_21670751_1 . . . 11
1.781 so_2603312_2603349_1 . . . 11
1.782 so_21611215_21811516_1 . . . 11
1.783 so_21611215_21811516_0 . . . 11
1.784 so_10778627_10778627_3 . . . 11
1.785 so_10778627_10778627_2 . . . 11
1.786 so_30267421_30267466_1 . . . 11
1.787 so_30267421_30267466_2 . . . 11
1.788 so_13443532_13445881_0 . . . 11
1.789 so_17400137_17400137_2 . . . 11
1.790 so_17400137_17400137_5 . . . 11
1.791 so_17400137_17403467_1 . . . 11
1.792 so_34191903_34192129_0 . . . 11
1.793 so_24194076_24218798_2 . . . 11
1.794 so_24194076_24194590_8 . . . 11
1.795 so_24194076_24206844_3 . . . 11
1.796 so_24194076_24196478_3 . . . 11
1.797 so_22817517_22817633_2 . . . 11
1.798 so_38501587_46472497_1 . . . 11
1.799 so_19861927_19862240_1 . . . 11
1.800 so_8885612_8885762_0 . . . 11
1.801 so_34005908_34006150_2 . . . 11
1.802 so_598148_599564_0 . . . 11
1.803 so_598148_599564_1 . . . 11

1.804 so_598148_598150_0 . . . 11
1.805 so_598148_598150_1 . . . 11
1.806 so_42997644_42998461_3 . . . 11
1.807 so_42997644_42998461_5 . . . 11
1.808 so_42997644_42999312_2 . . . 11
1.809 so_34860809_34932967_0 . . . 11
1.810 so_34860809_34860809_0 . . . 11
1.811 so_34860809_34860809_1 . . . 11
1.812 so_25323202_25323426_2 . . . 11
1.813 so_19586893_19670459_0 . . . 11
1.814 so_43263631_43264234_1 . . . 11
1.815 so_43263631_43264234_2 . . . 11
1.816 so_43263631_43264234_0 . . . 11
1.817 so_13744507_13745174_2 . . . 11
1.818 so_13744507_13745174_1 . . . 11
1.819 so_13744507_13744507_0 . . . 11
1.820 so_45734074_45734548_0 . . . 11
1.821 so_15680489_15680881_0 . . . 11
1.822 so_15680489_15680881_1 . . . 11
1.823 so_21133717_21134503_2 . . . 11
1.824 so_21133717_21134503_0 . . . 11
1.825 so_21133717_21134503_1 . . . 11
1.826 so_35819580_35819612_0 . . . 11
1.827 so_44309451_44309451_0 . . . 11
1.828 so_30379661_30379661_4 . . . 11
1.829 so_30379661_30379661_1 . . . 11
1.830 so_30379661_30379661_3 . . . 11
1.831 so_4737344_4737668_0 . . . 11
1.832 so_19472550_44932594_0 . . . 11
1.833 so_26707897_26708137_1 . . . 11
1.834 so_4638364_4638392_0 . . . 11
1.835 so_7229234_7229381_0 . . . 11
1.836 so_29513572_29513843_3 . . . 11
1.837 so_29513572_29513843_4 . . . 11
1.838 so_29513572_29513843_0 . . . 11
1.839 so_19231779_19232158_0 . . . 11
1.840 so_24186681_24186681_1 . . . 11
1.841 so_24186681_24187757_0 . . . 11
1.842 so_17772464_17772778_0 . . . 11
1.843 so_24869510_24870003_0 . . . 11
1.844 so_5214611_5214681_1 . . . 11
1.845 so_18819570_18819629_2 . . . 11
1.846 so_44698989_44699242_0 . . . 11
1.847 so_21147853_21148021_0 . . . 11
1.848 so_11813692_22003520_2 . . . 11
1.849 so_27158812_27158813_1 . . . 11
1.850 so_27158812_27158813_2 . . . 11
1.851 so_8852300_8852339_0 . . . 11
1.852 so_24796670_24796821_0 . . . 11
1.853 so_2934904_2934965_1 . . . 11
1.854 so_7112282_40953210_1 . . . 11

1.855 so_7112282_7112338_0 . . . 11
1.856 so_7112282_7112409_3 . . . 11
1.857 so_7112282_7112409_2 . . . 11
1.858 so_7112282_7112501_0 . . . 11
1.859 Multi-threaded executions and data races . . . 12
1.860 so_16642027_16642126_0 . . . 12
1.861 so_46819473_46820063_5 . . . 12
1.862 so_20445051_20445634_2 . . . 12
1.863 so_1932311_1932371_0 . . . 12
1.864 so_42441683_42442198_2 . . . 12
1.865 so_40357447_40360142_0 . . . 12
1.866 so_19759983_19760125_1 . . . 12
1.867 so_19759983_19781528_0 . . . 12
1.868 so_36613587_36613757_0 . . . 12
1.869 so_13932525_13932580_0 . . . 12
1.870 so_42754244_42754244_1 . . . 12
1.871 so_31531037_31552232_0 . . . 13
1.872 so_39925092_39925402_1 . . . 13
1.873 so_27429310_27429614_1 . . . 13
1.874 so_3503008_3503163_0 . . . 13
1.875 so_43910170_43930343_0 . . . 13
1.876 so_41371714_41416686_3 . . . 13
1.877 so_14713649_14713838_1 . . . 13
1.878 so_45597688_45603488_0 . . . 13
1.879 so_28591370_28591924_0 . . . 13
1.880 so_16847929_16848001_1 . . . 13
1.881 so_38721485_38721608_3 . . . 13
1.882 so_43265448_43266676_1 . . . 13
1.883 so_15325441_15325722_3 . . . 13
1.884 so_23043251_23043357_1 . . . 13
1.885 so_24682518_24682612_0 . . . 13
1.886 so_23368274_32394808_0 . . . 13
1.887 so_45793411_45794804_9 . . . 13
1.888 so_27024904_27025260_0 . . . 13
1.889 so_35226128_35226482_5 . . . 13
1.890 so_14365595_14368416_1 . . . 13
1.891 so_14127379_14127380_1 . . . 13
1.892 so_31067653_31067961_0 . . . 13
1.893 so_13359582_13359813_1 . . . 13
1.894 so_30691135_30692887_9 . . . 13
1.895 so_30691135_30692887_10 . . . 13
1.896 so_24632864_24633015_0 . . . 13
1.897 so_45960387_45962130_1 . . . 13
1.898 so_4213523_4213523_6 . . . 13
1.899 so_15270432_15270432_0 . . . 13
1.900 so_35226128_35226186_0 . . . 13
1.901 so_35128267_35128552_2 . . . 13
1.902 so_35128267_35128552_1 . . . 13
1.903 so_45475241_45476880_1 . . . 13
1.904 so_45475241_45477313_0 . . . 13
1.905 so_45475241_45476880_0 . . . 13

1.906 so_30691135_30692887_3 . . . 13
1.907 so_30691135_30691135_1 . . . 13
1.908 so_30691135_30692887_6 . . . 13
1.909 so_30691135_30692887_7 . . . 13
1.910 so_30691135_30692887_2 . . . 13
1.911 so_46065770_46069339_0 . . . 13
1.912 so_27333311_27333661_0 . . . 13
1.913 so_45637962_45642915_2 . . . 14
1.914 so_45694459_45694459_1 . . . 14
1.915 so_41371714_41416686_5 . . . 14
1.916 so_45689529_45692735_0 . . . 14
1.917 so_30691135_30692887_11 . . . 14
1.918 so_45694459_46008290_0 . . . 14
1.919 so_45694459_45694459_0 . . . 14
1.920 so_45475241_45477313_1 . . . 14
1.921 so_15313296_15313944_2 . . . 14
1.922 so_2488515_2488553_2 . . . 14
1.923 so_2488515_2488553_3 . . . 14
1.924 so_16391540_16397575_8 . . . 14
1.925 so_16391540_16397575_7 . . . 14
1.926 so_16391540_16397575_6 . . . 14
1.927 so_16391540_16397575_4 . . . 14
1.928 so_16391540_16397575_3 . . . 14
1.929 so_15313296_15313944_6 . . . 14
1.930 so_15313296_15313944_4 . . . 14
1.931 so_15313296_15313944_5 . . . 14
1.932 so_17768718_17768718_0 . . . 14
1.933 so_40930541_40931104_1 . . . 14
1.934 so_32574459_32574835_1 . . . 14
1.935 so_38952761_38963036_2 . . . 14
1.936 so_38952761_38963036_1 . . . 14
1.937 so_38952761_38963036_5 . . . 15
1.938 so_38952761_38963036_4 . . . 15
1.939 so_27333311_27333661_1 . . . 15
1.940 so_25290359_25290974_1 . . . 16
1.941 so_18223161_18430772_1 . . . 16
1.942 so_30691135_30692887_12 . . . 16
1.943 so_27333311_27333661_2 . . . 16
1.944 so_45475241_45502571_0 . . . 16
1.945 so_14846494_14846494_0 . . . 16
1.946 so_27429310_27429614_0 . . . 16
1.947 so_34863148_34863709_6 . . . 16
1.948 so_34863148_34863709_9 . . . 16
1.949 so_34863148_34863709_8 . . . 16
1.950 so_34863148_34863709_7 . . . 16
1.951 so_3503008_3503163_1 . . . 16
1.952 so_43910170_43930343_1 . . . 16
1.953 so_41371714_41416686_8 . . . 16
1.954 so_14713649_14713838_0 . . . 16
1.955 so_23385769_23386002_0 . . . 16
1.956 so_15863421_15863504_6 . . . 16

1.957 so_36613587_36613757_1 . . . 16
1.958 so_36613587_36613757_2 . . . 16
1.959 so_28591370_28591924_1 . . . 16
1.960 so_28591370_28591924_2 . . . 16
1.961 so_16847929_16848001_0 . . . 16
1.962 so_38721485_38721608_2 . . . 16
1.963 so_39641843_39642112_0 . . . 16
1.964 so_43265448_43266676_3 . . . 16
1.965 so_38399181_38399280_4 . . . 16
1.966 so_38399181_38399280_2 . . . 16
1.967 so_38399181_38399280_3 . . . 16
1.968 so_15325441_15325722_2 . . . 16
1.969 so_35109419_35109419_0 . . . 16
1.970 so_45960387_45962130_0 . . . 16
1.971 so_9552133_12919036_9 . . . 16
1.972 so_23043251_23043357_0 . . . 16
1.973 so_24682518_24682612_1 . . . 16
1.974 so_32990177_32990225_0 . . . 16
1.975 so_38952761_38963036_3 . . . 16
1.976 so_45793411_45794804_11 . . . 16
1.977 so_23233395_23233483_0 . . . 16
1.978 so_27024904_27025260_2 . . . 16
1.979 so_27024904_27025260_1 . . . 16
1.980 so_35226128_35226482_7 . . . 16
1.981 so_35226128_35226186_1 . . . 16
1.982 so_14365595_14368416_0 . . . 16
1.983 so_20823911_20826020_0 . . . 16
1.984 so_14127379_14127380_2 . . . 16
1.985 so_31067653_31067961_1 . . . 16
1.986 so_42179740_42213312_1 . . . 16
1.987 so_36682669_36699933_0 . . . 16
1.988 so_45885048_45885631_0 . . . 16
1.989 so_24682518_24682612_2 . . . 16
1.990 so_36682669_36699933_1 . . . 17
1.991 so_36414959_36416452_1 . . . 17
1.992 so_28111666_28115927_0 . . . 17
1.993 so_3592557_3592557_1 . . . 17
1.994 so_3592557_3592656_0 . . . 17
1.995 so_3592557_3592557_4 . . . 17
1.996 so_29268092_29268268_0 . . . 17
1.997 so_29295427_29295427_0 . . . 17
1.998 so_22837126_22837313_0 . . . 17
1.999 so_36814199_36814227_4 . . . 17
1.1000 so_36814199_36814227_3 . . . 17
1.1001 so_29708242_29708537_1 . . . 17
1.1002 so_29708242_29708537_0 . . . 17
1.1003 so_43749985_43751330_0 . . . 17
1.1004 so_29084160_29084412_0 . . . 17
1.1005 Acknowledgments . . . 17

2 Lexical conventions **18**

2.1 Separate translation . . . 18
2.2 so_16506196_16506229_3 . . . 18
2.3 so_3469588_3469588_7 . . . 18
2.4 so_1106149_1106172_0 . . . 18
2.5 so_18166885_18166923_5 . . . 18
2.6 Phases of translation . . . 18
2.7 so_6463014_6463014_2 . . . 18
2.8 so_6463014_6463014_0 . . . 18
2.9 so_6463014_6463014_1 . . . 18
2.10 so_6463014_6472770_0 . . . 18
2.11 so_6463014_6463725_0 . . . 18
2.12 so_25664848_25672839_5 . . . 18
2.13 so_323419_324732_0 . . . 18
2.14 so_323419_324732_2 . . . 18
2.15 so_323419_324732_1 . . . 18
2.16 so_41460467_41460863_1 . . . 18
2.17 so_15645170_15647342_0 . . . 18
2.18 so_22893226_22893340_0 . . . 18
2.19 so_13793355_13793447_1 . . . 18
2.20 so_13793355_13793447_0 . . . 18
2.21 so_28103029_28130061_2 . . . 18
2.22 so_28103029_28130061_4 . . . 18
2.23 so_28103029_28130061_1 . . . 18
2.24 so_28103029_28130061_5 . . . 18
2.25 so_28103029_28130061_3 . . . 18
2.26 so_41670800_41672455_0 . . . 18
2.27 so_43956379_43956379_0 . . . 18
2.28 so_12801776_12801788_0 . . . 18
2.29 so_38721301_38721411_0 . . . 18
2.30 so_16350099_16350099_1 . . . 18
2.31 so_30438211_30438211_0 . . . 18
2.32 so_24841083_24841123_1 . . . 18
2.33 so_23712259_23712374_0 . . . 18
2.34 so_23712259_23712371_0 . . . 18
2.35 so_36496414_36497078_0 . . . 18
2.36 so_46578158_46578210_0 . . . 18
2.37 so_35374858_35374875_0 . . . 18
2.38 so_31673945_31674194_0 . . . 18
2.39 so_15539935_15539957_2 . . . 18
2.40 so_12903294_12903307_1 . . . 18
2.41 so_12903294_12903307_0 . . . 18
2.42 so_9892369_9892431_0 . . . 18
2.43 so_9892369_9892426_1 . . . 18
2.44 so_9892369_9892426_0 . . . 18
2.45 so_8021484_8021557_0 . . . 18
2.46 so_8021484_8021484_0 . . . 18
2.47 so_8965655_8965674_0 . . . 18
2.48 so_34541818_34541818_1 . . . 18
2.49 so_3142630_3142716_0 . . . 18
2.50 so_27031839_27032096_2 . . . 18
2.51 so_12712832_12712833_0 . . . 18

2.52 so_38128836_38128863_1 . . . 18
2.53 so_39885423_39886017_3 . . . 18
2.54 so_39885423_39885473_0 . . . 18
2.55 so_45990793_45990884_0 . . . 18
2.56 so_45990793_45990884_1 . . . 18
2.57 so_45990793_45990793_0 . . . 18
2.58 so_12573816_12573818_0 . . . 18
2.59 so_12573816_12573818_1 . . . 18
2.60 so_5508110_5508168_0 . . . 18
2.61 so_37684327_37686206_0 . . . 18
2.62 so_37684327_37686206_1 . . . 18
2.63 so_37633691_37633859_0 . . . 18
2.64 so_30997129_30997287_2 . . . 18
2.65 so_21593_32064944_8 . . . 18
2.66 so_21593_77092_7 . . . 18
2.67 so_21593_77092_8 . . . 18
2.68 so_21593_32064944_7 . . . 18
2.69 so_19374878_19374879_0 . . . 18
2.70 so_3768363_4248640_2 . . . 18
2.71 so_25157614_25157694_0 . . . 18
2.72 so_25157614_25157729_0 . . . 18
2.73 so_25157614_25157708_1 . . . 18
2.74 so_25157614_25157708_0 . . . 18
2.75 so_25157614_25157770_0 . . . 18
2.76 so_18050661_18050895_0 . . . 18
2.77 so_331690_331935_0 . . . 18
2.78 so_18972735_18972735_3 . . . 18
2.79 so_18972735_18972735_0 . . . 18
2.80 so_18972735_18972735_1 . . . 18
2.81 so_18972735_18972735_2 . . . 18
2.82 so_32479995_32480112_0 . . . 18
2.83 so_14242504_14242532_1 . . . 18
2.84 so_14242504_14242532_0 . . . 18
2.85 so_14242504_14242526_0 . . . 18
2.86 so_30153902_30153902_0 . . . 18
2.87 so_15311447_15311928_0 . . . 18
2.88 so_15311447_15311447_0 . . . 18
2.89 so_11869593_11869665_1 . . . 18
2.90 so_31742318_31742664_0 . . . 18
2.91 so_19357027_19436896_0 . . . 18
2.92 so_11721981_11722479_1 . . . 18
2.93 so_4385297_4385297_6 . . . 18
2.94 so_38197026_38197171_0 . . . 18
2.95 so_39548465_39625214_3 . . . 18
2.96 so_12305868_12307147_0 . . . 18
2.97 so_8192977_8193471_0 . . . 18
2.98 so_5708259_5708403_0 . . . 18
2.99 so_5708259_26127812_4 . . . 18
2.100 so_5708259_26127812_3 . . . 18
2.101 so_5708259_5708581_0 . . . 18
2.102 so_12574093_12574099_0 . . . 18

2.103 so_28821786_28821799_2 . . . 18
2.104 so_28821786_28821799_3 . . . 18
2.105 so_28821786_28821799_1 . . . 18
2.106 so_28821786_28821799_5 . . . 18
2.107 so_28821786_28821799_0 . . . 18
2.108 so_28821786_28821799_4 . . . 18
2.109 so_28900842_28900986_0 . . . 18
2.110 so_28900842_28900986_2 . . . 18
2.111 so_24469809_24470424_2 . . . 18
2.112 so_2837469_2837506_0 . . . 18
2.113 so_2837469_2837499_0 . . . 18
2.114 so_2837469_2837506_1 . . . 18
2.115 so_6794590_6795706_1 . . . 18
2.116 so_6794590_6795706_2 . . . 18
2.117 so_6794590_6795706_3 . . . 18
2.118 so_1819131_1825872_2 . . . 18
2.119 so_17442912_17443004_0 . . . 18
2.120 so_43824729_43824729_0 . . . 18
2.121 so_8833524_8834196_1 . . . 18
2.122 so_8833524_8834196_2 . . . 18
2.123 so_21629073_21629191_0 . . . 18
2.124 so_25605783_25605783_1 . . . 18
2.125 so_6855149_9651803_0 . . . 18
2.126 so_5486669_5486724_0 . . . 18
2.127 so_5486669_5486724_1 . . . 18
2.128 so_23856294_23856342_0 . . . 18
2.129 so_14969771_14969771_0 . . . 18
2.130 so_14969771_14970757_0 . . . 18
2.131 so_13754778_13769502_1 . . . 18
2.132 so_13754778_13758060_3 . . . 18
2.133 so_13754778_13758060_1 . . . 18
2.134 so_36761617_36761692_0 . . . 18
2.135 so_40563538_40563546_0 . . . 18
2.136 so_5106280_27485711_2 . . . 18
2.137 so_5106280_27485711_1 . . . 18
2.138 so_5106280_27485711_0 . . . 18
2.139 so_5106280_27485711_3 . . . 18
2.140 so_25468785_25468848_0 . . . 18
2.141 so_72271_31850899_0 . . . 18
2.142 so_72271_8172348_0 . . . 18
2.143 so_72271_72409_0 . . . 18
2.144 so_72271_72450_0 . . . 18
2.145 so_7210286_7901567_0 . . . 18
2.146 so_7210286_7901567_1 . . . 18
2.147 so_18040393_32413257_3 . . . 18
2.148 so_42138264_42138264_0 . . . 18
2.149 so_12305240_12305397_0 . . . 18
2.150 so_5676978_5676978_1 . . . 18
2.151 so_32154832_32157764_1 . . . 18
2.152 so_22977043_24067054_0 . . . 18
2.153 so_15445593_15445593_0 . . . 18

2.154 Character sets . . . 19
2.155 so_18494262_18494321_5 . . . 19
2.156 so_18494262_18494321_4 . . . 19
2.157 so_27927874_42655380_1 . . . 19
2.158 so_18866318_18866362_5 . . . 19
2.159 so_18866318_18866362_0 . . . 19
2.160 so_15057233_15057271_0 . . . 19
2.161 so_36496414_36497078_1 . . . 19
2.162 so_35977940_35977999_0 . . . 19
2.163 so_27872517_27872518_1 . . . 19
2.164 so_40563390_40569809_1 . . . 19
2.165 so_27031839_27032096_4 . . . 19
2.166 so_14595300_14595365_0 . . . 19
2.167 so_21364673_21364673_6 . . . 19
2.168 so_9416926_9416926_0 . . . 19
2.169 so_28966141_28966680_0 . . . 19
2.170 so_28900842_28900986_1 . . . 19
2.171 so_28900842_28900986_3 . . . 19
2.172 so_10938306_10938350_0 . . . 19
2.173 so_46540621_46541819_1 . . . 19
2.174 so_22940011_22940529_0 . . . 19
2.175 so_12573805_12573805_0 . . . 20
2.176 so_15311447_15311928_1 . . . 20
2.177 so_7081367_7081484_3 . . . 20
2.178 so_28966141_28966680_5 . . . 20
2.179 so_28966141_28966680_6 . . . 20
2.180 so_24469809_24469850_9 . . . 20
2.181 so_2462624_2462788_1 . . . 20
2.182 so_11496033_11496120_0 . . . 20
2.183 so_22893226_22893340_1 . . . 20
2.184 so_19727894_19727922_1 . . . 20
2.185 so_25756143_25756255_0 . . . 20
2.186 so_30063956_30064005_0 . . . 20
2.187 so_15598698_15598721_0 . . . 20
2.188 so_15598698_15598759_5 . . . 20
2.189 so_9220838_9220852_0 . . . 20
2.190 so_27872517_27872518_2 . . . 20
2.191 so_11138188_11138188_4 . . . 20
2.192 so_37953843_37955020_1 . . . 20
2.193 so_3768363_3768363_0 . . . 20
2.194 so_3768363_4248640_1 . . . 20
2.195 so_13454183_13454256_0 . . . 20
2.196 so_7081367_7081484_2 . . . 20
2.197 so_21364673_21364673_8 . . . 20
2.198 so_9416926_9416946_0 . . . 20
2.199 so_42632224_42632606_2 . . . 20
2.200 so_24114365_45938074_0 . . . 20
2.201 so_17320769_17320787_3 . . . 20
2.202 so_17320769_17320787_2 . . . 20
2.203 so_29424726_29424770_0 . . . 20
2.204 so_32586834_32587003_1 . . . 20

2.205 so_32586834_32587003_0 . . . 20
2.206 so_2503269_2503303_0 . . . 20
2.207 so_29394518_29394804_5 . . . 20
2.208 so_29394518_29394804_9 . . . 20
2.209 so_29394518_29394804_8 . . . 20
2.210 so_40537750_40538005_2 . . . 20
2.211 Trigraph sequences . . . 20
2.212 so_17442912_17442949_0 . . . 20
2.213 so_17718286_17718330_0 . . . 20
2.214 Preprocessing tokens . . . 21
2.215 so_11074073_11074102_2 . . . 21
2.216 so_14763892_14763892_0 . . . 21
2.217 so_5486669_5486724_2 . . . 21
2.218 so_1206624_1206939_0 . . . 21
2.219 so_39885423_39886017_2 . . . 21
2.220 so_39885423_39886017_5 . . . 21
2.221 so_39885423_39886017_4 . . . 21
2.222 so_13050234_21791272_0 . . . 21
2.223 so_33670238_33670239_0 . . . 21
2.224 so_14882730_14882925_1 . . . 21
2.225 so_6855149_9651803_1 . . . 21
2.226 so_7068260_7068363_0 . . . 21
2.227 so_7068260_7068363_1 . . . 21
2.228 so_28354108_28354898_0 . . . 21
2.229 so_28354108_28354898_1 . . . 21
2.230 so_30308088_30308215_0 . . . 21
2.231 so_13754778_13769502_12 . . . 21
2.232 so_13754778_13769502_10 . . . 21
2.233 so_13754778_13769502_11 . . . 21
2.234 so_13754778_13769502_13 . . . 21
2.235 so_8338121_8338187_0 . . . 21
2.236 Alternative tokens . . . 22
2.237 so_17110294_17110423_1 . . . 22
2.238 so_13269513_13270173_1 . . . 22
2.239 so_4251699_4251799_2 . . . 22
2.240 so_4251699_4251799_0 . . . 22
2.241 so_17110294_17110423_2 . . . 22
2.242 so_13269513_13270173_2 . . . 22
2.243 so_4251699_4251799_3 . . . 22
2.244 so_4251699_4251799_1 . . . 22
2.245 so_4251699_4251718_1 . . . 22
2.246 so_34541818_34541818_0 . . . 22
2.247 Tokens . . . 22
2.248 so_25452027_25452099_0 . . . 22
2.249 so_25452027_25452161_0 . . . 22
2.250 so_37240340_37240429_0 . . . 22
2.251 so_13754778_13754916_0 . . . 22
2.252 Comments . . . 22
2.253 so_30438211_30438211_1 . . . 22
2.254 so_41909225_41910370_0 . . . 22
2.255 so_32190239_32190386_0 . . . 22

2.256 so_41800617_41800688_0 . . . 22
2.257 Header names . . . 23
2.258 so_15014593_15015062_0 . . . 23
2.259 so_34692366_34692862_0 . . . 23
2.260 so_36290995_36292219_3 . . . 23
2.261 so_5790161_5790259_2 . . . 23
2.262 so_5790161_5790259_5 . . . 23
2.263 so_5790161_5790259_4 . . . 23
2.264 so_29119638_29119900_0 . . . 23
2.265 so_29119638_29119900_1 . . . 23
2.266 Preprocessing numbers . . . 23
2.267 so_24147150_24148306_0 . . . 23
2.268 Identifiers . . . 23
2.269 so_42275543_42275617_0 . . . 24
2.270 so_32799078_32799263_0 . . . 24
2.271 so_9238457_9239810_0 . . . 24
2.272 so_19790451_19790562_3 . . . 24
2.273 so_5676978_5676978_0 . . . 24
2.274 so_37901693_37901693_3 . . . 24
2.275 so_31440790_31440790_0 . . . 24
2.276 so_30311551_30311551_0 . . . 24
2.277 so_28163187_28163251_6 . . . 24
2.278 so_30404388_30404388_0 . . . 24
2.279 so_27777071_27777306_1 . . . 24
2.280 so_33856187_33856325_0 . . . 24
2.281 so_33856187_33856407_1 . . . 24
2.282 so_29971760_29971760_0 . . . 24
2.283 so_47120549_47120620_0 . . . 24
2.284 so_22591490_22591837_1 . . . 24
2.285 so_36547516_36552661_4 . . . 24
2.286 so_26301737_26301737_2 . . . 24
2.287 Keywords . . . 24
2.288 so_6399615_6402166_0 . . . 24
2.289 so_18578025_18578076_1 . . . 24
2.290 so_13661328_13661361_0 . . . 24
2.291 so_36547516_36552661_0 . . . 24
2.292 so_2419805_2419867_0 . . . 24
2.293 so_36547516_36552661_2 . . . 24
2.294 Operators and punctuators . . . 24
2.295 so_6464436_6466219_0 . . . 24
2.296 so_25452027_25452161_3 . . . 24
2.297 so_25452027_25452161_1 . . . 24
2.298 so_28354108_28354898_2 . . . 24
2.299 Literals . . . 25
2.300 so_22198779_22199042_1 . . . 25
2.301 so_31163449_31163665_1 . . . 26
2.302 so_9670854_42998291_1 . . . 26
2.303 so_35374858_35374875_1 . . . 26
2.304 so_27551167_27551413_0 . . . 26
2.305 so_26568200_26568224_0 . . . 26
2.306 so_14695118_14695400_2 . . . 26

2.307 so_2740715_2740721_0 . . . 26
2.308 so_7439268_7439387_0 . . . 26
2.309 so_12786579_12786656_0 . . . 26
2.310 so_31150380_31150380_0 . . . 26
2.311 so_45469214_45469321_2 . . . 26
2.312 so_39050556_39050571_0 . . . 26
2.313 so_33095999_33100488_1 . . . 26
2.314 so_30448926_30449001_0 . . . 26
2.315 so_12553173_12553186_1 . . . 26
2.316 so_45982485_45983300_0 . . . 27
2.317 so_11931452_11931607_0 . . . 27
2.318 so_36560143_36560446_1 . . . 27
2.319 so_23445662_23445826_0 . . . 27
2.320 so_15575054_15575107_0 . . . 27
2.321 so_15769226_15769395_0 . . . 27
2.322 so_8111357_8111507_1 . . . 27
2.323 so_40319789_40319850_0 . . . 27
2.324 so_19073325_19073408_0 . . . 27
2.325 so_42459750_42497336_0 . . . 27
2.326 so_14695118_14695400_3 . . . 27
2.327 so_4434140_4690988_1 . . . 27
2.328 so_4434140_4690988_2 . . . 27
2.329 so_30726423_30729298_0 . . . 27
2.330 so_23958490_23959049_0 . . . 27
2.331 so_42762610_42762645_0 . . . 27
2.332 so_25633040_25633075_0 . . . 27
2.333 so_27405418_27405419_1 . . . 27
2.334 so_5563000_5563063_7 . . . 27
2.335 so_23687534_23687929_0 . . . 27
2.336 so_12786579_12786656_1 . . . 27
2.337 so_31150380_31150380_1 . . . 27
2.338 so_18706859_18707097_0 . . . 27
2.339 so_18706859_18707097_1 . . . 27
2.340 so_19167208_19167928_0 . . . 27
2.341 so_19167208_19167928_1 . . . 27
2.342 so_20727531_20727551_0 . . . 27
2.343 so_15744310_15744362_0 . . . 27
2.344 so_8108642_8108658_2 . . . 27
2.345 so_8108642_8108658_0 . . . 27
2.346 so_8108642_8108715_0 . . . 27
2.347 so_33095999_33100488_2 . . . 27
2.348 so_33095999_33100488_3 . . . 27
2.349 so_15481532_15481828_0 . . . 27
2.350 so_24147150_24148306_1 . . . 27
2.351 so_15805394_15805394_2 . . . 27
2.352 so_8111357_8111507_2 . . . 27
2.353 so_14695118_14695400_4 . . . 27
2.354 so_18706859_18707097_2 . . . 27
2.355 so_33571554_33573507_2 . . . 27
2.356 so_8108642_8108658_1 . . . 27
2.357 so_19394141_19394188_0 . . . 28

2.358 so_27081797_27081934_1 . . . 28
2.359 so_13547368_13547368_1 . . . 28
2.360 so_13547368_13554733_2 . . . 28
2.361 so_13547368_13547368_0 . . . 28
2.362 so_13547368_13554733_0 . . . 28
2.363 so_23884829_23884869_1 . . . 28
2.364 so_22893226_22893340_2 . . . 28
2.365 so_16944750_17129670_0 . . . 28
2.366 so_45093121_45093191_0 . . . 28
2.367 so_15057233_15058572_0 . . . 28
2.368 so_15057233_15057233_0 . . . 28
2.369 so_3960954_34571914_0 . . . 28
2.370 so_47125609_47125647_0 . . . 28
2.371 so_2172943_47052994_0 . . . 28
2.372 so_32868769_32868947_0 . . . 28
2.373 so_25375274_25376396_0 . . . 28
2.374 so_19906802_19906869_1 . . . 28
2.375 so_30035547_30035811_1 . . . 28
2.376 so_15598698_15598759_0 . . . 28
2.377 so_1826464_1826484_0 . . . 28
2.378 so_23066901_23066995_0 . . . 28
2.379 so_23066901_23066981_4 . . . 28
2.380 so_699301_699323_0 . . . 28
2.381 so_699301_699323_1 . . . 28
2.382 so_36496013_36496102_0 . . . 28
2.383 so_31743001_31743285_0 . . . 28
2.384 so_6149314_6149413_0 . . . 28
2.385 so_6149314_6149365_0 . . . 28
2.386 so_7459939_7469242_0 . . . 28
2.387 so_7459939_7459943_1 . . . 28
2.388 so_40436990_40437223_0 . . . 28
2.389 so_28160535_28160671_1 . . . 28
2.390 so_12641908_12642132_1 . . . 28
2.391 so_29101247_29101390_0 . . . 28
2.392 so_27614228_27614228_1 . . . 28
2.393 so_22049889_22050083_0 . . . 28
2.394 so_22353364_22353505_1 . . . 28
2.395 so_13509040_13509341_0 . . . 28
2.396 so_17082512_17082756_0 . . . 28
2.397 so_32106336_32106396_0 . . . 28
2.398 so_1635130_1635140_2 . . . 28
2.399 so_1635130_1635140_0 . . . 28
2.400 so_1635130_1635140_3 . . . 28
2.401 so_1635130_1635140_1 . . . 28
2.402 so_27590281_27590339_0 . . . 28
2.403 so_6440812_6440946_0 . . . 28
2.404 so_6440812_6440983_1 . . . 28
2.405 so_6440812_6440812_0 . . . 28
2.406 so_13554914_13554914_0 . . . 28
2.407 so_22776619_22777509_0 . . . 28
2.408 so_39548465_39552788_0 . . . 28

2.409 so_10938306_30872695_2 . . . 28
2.410 so_25693182_25694756_0 . . . 28
2.411 so_11241313_11241313_0 . . . 28
2.412 so_19374878_19374879_3 . . . 28
2.413 so_19374878_19374879_2 . . . 28
2.414 so_26062865_26064070_0 . . . 28
2.415 so_32268181_32268299_0 . . . 28
2.416 so_20739884_20739968_0 . . . 28
2.417 so_13547368_13554733_3 . . . 29
2.418 so_15057233_15057233_1 . . . 29
2.419 so_15057233_15057583_0 . . . 29
2.420 so_35492365_35492552_1 . . . 29
2.421 so_13554914_13554914_1 . . . 29
2.422 so_31239524_31239648_0 . . . 29
2.423 so_4705968_4706217_3 . . . 29
2.424 so_2735101_2735111_0 . . . 29
2.425 so_35152948_35152948_1 . . . 29
2.426 so_13547368_13554733_1 . . . 29
2.427 so_28351017_28351453_1 . . . 29
2.428 so_28351017_28351114_2 . . . 29
2.429 so_16350099_16350099_2 . . . 29
2.430 so_13454183_13454256_1 . . . 29
2.431 so_7081367_7081484_1 . . . 29
2.432 so_21785095_21785191_1 . . . 30
2.433 so_33396728_33398483_1 . . . 30
2.434 so_30003519_30003618_0 . . . 30
2.435 so_4353780_4353788_0 . . . 30
2.436 so_12205141_12205279_0 . . . 30
2.437 so_9834273_9834608_0 . . . 30
2.438 so_31823523_31823653_0 . . . 30
2.439 so_31823523_31823653_1 . . . 30
2.440 so_25229832_25231230_0 . . . 30
2.441 so_24971431_24971503_0 . . . 30
2.442 so_17481015_17481021_0 . . . 30
2.443 so_8757815_8758072_0 . . . 30
2.444 so_33085035_33085226_0 . . . 30
2.445 so_36017229_36017726_0 . . . 30
2.446 so_3213445_3213479_0 . . . 30
2.447 so_46810474_46810797_0 . . . 30
2.448 so_21526752_39453523_6 . . . 30
2.449 so_12553359_12553405_0 . . . 30
2.450 so_25283526_25284192_0 . . . 30
2.451 so_28635257_28635396_0 . . . 30
2.452 so_39108471_39109030_1 . . . 30
2.453 so_15911714_16006093_1 . . . 30
2.454 so_22723822_22723901_1 . . . 31
2.455 so_34917349_34917464_1 . . . 31
2.456 so_32519993_32520133_0 . . . 31
2.457 so_28114301_28117559_0 . . . 31
2.458 so_34917349_34917464_2 . . . 31
2.459 so_31818299_31818825_1 . . . 31

2.460 so_31818299_31818600_0 . . . 31
2.461 so_46540621_46541819_2 . . . 31
2.462 so_39885423_39885551_0 . . . 31
2.463 so_39885423_39886017_0 . . . 31
2.464 so_3093632_7366469_0 . . . 31
2.465 so_34917349_34917464_3 . . . 31
2.466 so_22668264_22668324_0 . . . 31
2.467 so_22893226_22893340_3 . . . 31
2.468 so_36308071_36308300_0 . . . 31
2.469 so_35765449_35765500_2 . . . 31
2.470 so_25886998_25887239_0 . . . 31
2.471 so_7424647_7424647_0 . . . 31
2.472 so_32949403_32949464_1 . . . 31
2.473 so_32995010_32995212_0 . . . 31
2.474 so_14903883_14903968_0 . . . 31
2.475 so_21529194_21529283_0 . . . 31
2.476 so_3303164_3303180_0 . . . 31
2.477 so_15598023_15598100_0 . . . 31
2.478 so_27375117_27375117_0 . . . 31
2.479 so_33692872_33692987_1 . . . 31
2.480 so_6949150_6949288_0 . . . 31
2.481 so_18149191_18149441_1 . . . 31
2.482 so_2033110_2033395_0 . . . 31
2.483 so_26972373_26972496_1 . . . 31
2.484 so_26972373_26972528_0 . . . 31
2.485 so_15508148_15508172_0 . . . 31
2.486 so_15508148_15508173_0 . . . 31
2.487 so_24760041_24760066_0 . . . 31
2.488 so_17471497_17471602_5 . . . 31
2.489 so_27450021_27451656_0 . . . 31
2.490 so_32089214_32089365_0 . . . 31
2.491 so_21844064_21844136_0 . . . 31
2.492 so_21844064_21844102_1 . . . 31
2.493 so_24873891_24874210_0 . . . 31
2.494 so_23145793_23799618_1 . . . 31
2.495 so_14034827_14034835_0 . . . 31
2.496 so_38294292_38295071_4 . . . 31
2.497 so_3607974_3608062_0 . . . 31
2.498 so_13275162_13275224_3 . . . 31
2.499 so_44888502_44889285_1 . . . 31
2.500 so_16776444_16776444_2 . . . 31
2.501 so_34713756_34713858_2 . . . 31
2.502 so_41435938_41435938_1 . . . 31
2.503 so_349025_349031_0 . . . 31
2.504 so_1588976_1589073_0 . . . 31
2.505 so_1588976_1589014_0 . . . 31
2.506 so_27275635_27275713_1 . . . 31
2.507 so_45056234_45056630_0 . . . 31
2.508 so_14415488_14415488_1 . . . 31
2.509 so_34560711_34560743_0 . . . 31
2.510 so_15924677_15924726_0 . . . 31

2.511 so_12016757_12016757_0 . . . 31
2.512 so_32519993_32520133_1 . . . 31
2.513 so_30888851_30891329_1 . . . 31
2.514 so_3047567_3047593_0 . . . 31
2.515 so_18325501_18325583_1 . . . 31
2.516 so_18423201_18423229_0 . . . 31
2.517 so_38297535_38297654_0 . . . 31
2.518 so_2327841_2327980_0 . . . 31
2.519 so_33012680_33012777_1 . . . 31
2.520 so_8171337_8171388_0 . . . 31
2.521 so_30293262_30293262_0 . . . 31
2.522 so_15943514_15943544_1 . . . 31
2.523 so_18819849_18819935_1 . . . 31
2.524 so_17474090_17474090_0 . . . 31
2.525 so_13275162_13275224_0 . . . 31
2.526 so_22776619_22777509_1 . . . 31
2.527 so_18325501_18325583_0 . . . 31
2.528 so_32652080_32652348_1 . . . 31
2.529 so_6628908_6629002_1 . . . 31
2.530 so_2985622_2985635_0 . . . 32
2.531 so_5206957_5206967_0 . . . 32
2.532 so_22990044_22990174_0 . . . 32
2.533 so_38294292_38295071_2 . . . 32
2.534 so_4383946_4383946_1 . . . 32
2.535 so_4383946_4383946_4 . . . 32
2.536 so_13087219_13087285_1 . . . 32
2.537 so_18724889_18724993_0 . . . 32
2.538 so_18325501_18325583_2 . . . 32
2.539 so_13793294_13793331_0 . . . 32
2.540 so_14664510_14664525_0 . . . 32
2.541 so_13515561_13515903_0 . . . 32
2.542 so_21941153_21941210_0 . . . 32
2.543 so_14903883_14903968_1 . . . 32
2.544 so_21529194_21529283_2 . . . 32
2.545 so_3287135_3287200_1 . . . 32
2.546 so_10742335_10742372_1 . . . 32
2.547 so_6949150_6949202_0 . . . 32
2.548 so_47492659_47492689_1 . . . 32
2.549 so_47492659_47492689_0 . . . 32
2.550 so_26399643_26399769_0 . . . 32
2.551 so_27450021_27450079_0 . . . 32
2.552 so_23145793_23799618_3 . . . 32
2.553 so_38294292_38295071_5 . . . 32
2.554 so_1870329_1870355_1 . . . 32
2.555 so_16459721_16460071_0 . . . 32
2.556 so_17909417_17909443_1 . . . 32
2.557 so_6380666_6380755_0 . . . 32
2.558 so_17425372_17425429_0 . . . 32
2.559 so_46600017_46601362_1 . . . 32
2.560 so_19186793_19186884_0 . . . 32
2.561 so_16019478_16019528_0 . . . 32

2.562 so_19372952_19372981_0 . . . 32
2.563 so_21182296_21183844_0 . . . 32
2.564 so_26279628_26279717_1 . . . 32
2.565 so_26279628_26279904_0 . . . 32
2.566 so_14468460_14468561_0 . . . 32
2.567 so_35490327_35490471_0 . . . 32
2.568 so_1611673_1611689_0 . . . 32
2.569 so_2397984_4105123_0 . . . 32
2.570 so_6628908_6629002_5 . . . 32
2.571 so_23482297_23483028_0 . . . 32
2.572 so_21788652_21789691_1 . . . 32
2.573 so_41460467_41460863_0 . . . 32
2.574 so_29632929_29634022_1 . . . 32
2.575 so_30186249_30186270_2 . . . 32
2.576 so_9892369_9892431_1 . . . 32
2.577 so_5784711_5784933_0 . . . 32
2.578 so_8965655_8965674_1 . . . 32
2.579 so_5179384_5182960_1 . . . 32
2.580 so_5179384_5182960_0 . . . 32
2.581 so_5179384_5179432_0 . . . 32
2.582 so_20687283_20687603_0 . . . 32
2.583 so_20687283_20687576_1 . . . 32
2.584 so_20687283_20687576_0 . . . 32
2.585 so_39327148_39327210_0 . . . 32
2.586 so_27909861_27910289_0 . . . 32
2.587 so_8192977_8193471_1 . . . 32
2.588 so_31809141_31809585_1 . . . 32
2.589 so_31809141_31809585_0 . . . 32
2.590 so_44279873_44279873_1 . . . 32
2.591 so_32949403_32949464_2 . . . 32
2.592 so_15111395_15111434_0 . . . 32
2.593 so_30888851_30891329_2 . . . 32
2.594 so_38297535_38297654_1 . . . 32
2.595 so_28351017_28351114_1 . . . 32
2.596 so_28351017_28351114_0 . . . 32
2.597 so_21844064_21844102_2 . . . 32
2.598 so_4383946_4383946_5 . . . 32
2.599 so_4383946_4383946_2 . . . 32
2.600 so_4383946_4383946_6 . . . 32
2.601 so_22776619_22776619_0 . . . 32
2.602 so_39548465_39625214_2 . . . 32
2.603 so_39548465_39552788_1 . . . 32
2.604 so_28966141_28966680_7 . . . 32
2.605 so_36761617_36761957_2 . . . 32
2.606 so_33380742_33398698_8 . . . 33
2.607 so_38297100_38297115_1 . . . 33
2.608 so_34901903_34902455_0 . . . 33
2.609 so_28696522_28696670_0 . . . 33
2.610 so_1282295_1282328_1 . . . 33
2.611 so_15182240_15182308_0 . . . 33
2.612 so_13091804_13091820_0 . . . 33

2.613 so_24212792_24212940_3 . . . 33
2.614 so_18775044_18775126_1 . . . 33
2.615 so_38297100_38297115_0 . . . 33
2.616 so_27121395_27121512_0 . . . 33
2.617 so_5547664_5547677_1 . . . 33
2.618 so_23564044_23564720_5 . . . 33
2.619 so_20405569_20405975_0 . . . 33
2.620 so_31036153_31036258_2 . . . 33
2.621 so_47278515_47278557_0 . . . 33
2.622 so_14189709_14189818_1 . . . 33
2.623 so_5380358_5684246_0 . . . 33
2.624 so_32250408_32250549_0 . . . 33
2.625 so_8100519_8100583_0 . . . 33
2.626 so_25452245_25452626_0 . . . 33
2.627 so_14915238_14920864_3 . . . 33
2.628 so_5684253_5684253_0 . . . 33
2.629 so_32250408_32250549_1 . . . 33
2.630 so_25875228_25875340_0 . . . 33
2.631 so_2111667_2112101_0 . . . 34
2.632 so_40576175_40576294_1 . . . 34
2.633 so_40576175_40576294_2 . . . 34
2.634 so_4385297_4385297_2 . . . 34
2.635 so_4385297_4385297_0 . . . 34
2.636 so_4385297_4385297_4 . . . 34
2.637 so_14505971_14506014_0 . . . 34
2.638 so_13112369_13112369_1 . . . 34
2.639 so_10880507_10880584_0 . . . 34
2.640 so_14599754_14601372_0 . . . 34
2.641 so_8097767_8098054_0 . . . 34
2.642 so_32652080_32652348_0 . . . 34

3 Basic concepts **36**
3.1 so_23440407_23440407_0 . . . 36
3.2 so_21655523_21655523_0 . . . 36
3.3 so_23974070_23979947_0 . . . 36
3.4 so_21662586_21662586_0 . . . 36
3.5 so_14270941_14271402_3 . . . 36
3.6 so_41290827_41290827_2 . . . 36
3.7 so_31234128_31234128_0 . . . 36
3.8 so_31708705_31710965_0 . . . 36
3.9 so_46020637_46020924_0 . . . 36
3.10 so_46020637_46025291_1 . . . 36
3.11 so_23988716_23988716_0 . . . 36
3.12 so_24168137_24168426_3 . . . 36
3.13 so_12721584_12721722_0 . . . 36
3.14 so_46020637_46025291_2 . . . 36
3.15 so_34696351_34732843_10 . . . 36
3.16 so_610245_613132_4 . . . 36
3.17 so_31234128_31234415_0 . . . 36
3.18 so_46020637_46025291_3 . . . 36
3.19 so_23988716_23988716_1 . . . 36

3.20 so_31440790_31440790_1 . . . 36
3.21 so_33371684_33372052_0 . . . 36
3.22 so_26711536_26711765_1 . . . 36
3.23 so_23974070_23979947_1 . . . 36
3.24 so_40962505_40962746_1 . . . 36
3.25 so_12987259_12987259_0 . . . 36
3.26 so_16251127_16253667_0 . . . 36
3.27 so_41760241_41762123_1 . . . 36
3.28 so_42053409_42053409_2 . . . 36
3.29 so_41148054_41195598_1 . . . 36
3.30 so_41148054_41148054_1 . . . 36
3.31 so_25853258_25853661_0 . . . 36
3.32 so_23988716_31234708_0 . . . 36
3.33 so_23988716_23988716_2 . . . 36
3.34 so_610245_613132_0 . . . 36
3.35 so_24107629_24107763_2 . . . 36
3.36 so_24107629_24107763_4 . . . 36
3.37 so_24107629_24107763_3 . . . 36
3.38 so_24107629_24107763_1 . . . 36
3.39 so_24107629_24107763_5 . . . 36
3.40 so_27221737_27221737_1 . . . 36
3.41 so_27221737_27221737_2 . . . 36
3.42 so_26529842_26530462_1 . . . 36
3.43 Declarations and definitions . . . 37
3.44 so_23345554_23345856_0 . . . 37
3.45 so_5495936_5495936_0 . . . 37
3.46 so_5495936_5495936_1 . . . 37
3.47 so_34247975_34248633_6 . . . 37
3.48 so_32715610_32716199_0 . . . 37
3.49 so_1410563_1411005_0 . . . 37
3.50 so_10723892_10734635_1 . . . 37
3.51 so_17996729_17996786_4 . . . 37
3.52 so_44607634_45719873_1 . . . 37
3.53 so_24589816_24589826_1 . . . 37
3.54 so_24688557_24688652_0 . . . 37
3.55 so_26386010_26386337_0 . . . 37
3.56 so_28752247_28752372_0 . . . 37
3.57 so_16883333_16883496_1 . . . 37
3.58 so_46726390_46742542_3 . . . 37
3.59 so_29578253_29578355_0 . . . 37
3.60 so_23345554_23345856_1 . . . 37
3.61 so_31535401_31535603_0 . . . 37
3.62 so_12146211_12146246_0 . . . 37
3.63 so_34247975_34248633_10 . . . 37
3.64 so_34247975_34248633_7 . . . 37
3.65 so_24771192_24771554_0 . . . 37
3.66 so_4084972_4084996_0 . . . 37
3.67 so_18771299_18785180_0 . . . 37
3.68 so_6318779_6318850_0 . . . 37
3.69 so_26462279_26462780_0 . . . 37
3.70 so_17996729_17996786_5 . . . 37

3.71 so_23646847_23646922_0 . . . 37
3.72 so_23654246_23654434_0 . . . 37
3.73 so_11867431_11868086_0 . . . 37
3.74 so_24688557_24688652_1 . . . 37
3.75 so_33908822_33910578_0 . . . 37
3.76 so_5704914_5705232_1 . . . 37
3.77 so_3025997_3026237_1 . . . 37
3.78 so_43304332_43305321_0 . . . 37
3.79 so_30803399_30803988_0 . . . 37
3.80 so_8069717_8069831_0 . . . 37
3.81 so_7730874_7731106_2 . . . 37
3.82 so_34247975_34248633_9 . . . 37
3.83 so_33178981_33179101_2 . . . 37
3.84 so_13016065_13016273_3 . . . 37
3.85 so_13016065_13016273_2 . . . 37
3.86 so_28526585_28527093_3 . . . 37
3.87 so_28526585_28527093_4 . . . 37
3.88 so_40257011_40257321_1 . . . 38
3.89 so_31875710_31876176_0 . . . 38
3.90 so_9929703_9929734_2 . . . 38
3.91 One definition rule . . . 38
3.92 so_12978745_12978745_1 . . . 38
3.93 so_16642027_16642148_0 . . . 38
3.94 so_42120053_42120158_0 . . . 38
3.95 so_14654377_14654428_0 . . . 38
3.96 so_22186751_22186816_1 . . . 38
3.97 so_40962505_40962746_0 . . . 38
3.98 so_15865634_15865729_2 . . . 38
3.99 so_1410563_1411005_1 . . . 38
3.100 so_29287361_29287458_0 . . . 38
3.101 so_45900040_45901227_0 . . . 38
3.102 so_23654246_23654434_2 . . . 38
3.103 so_8329871_8329930_0 . . . 38
3.104 so_5337370_5337391_1 . . . 38
3.105 so_4145915_4146296_0 . . . 38
3.106 so_36841401_36841449_1 . . . 38
3.107 so_18629184_18629454_1 . . . 38
3.108 so_25300461_25300461_0 . . . 38
3.109 so_23973281_23976394_0 . . . 38
3.110 so_4325802_4325865_0 . . . 38
3.111 so_23494193_23494320_0 . . . 38
3.112 so_25671850_25672020_0 . . . 38
3.113 so_18202324_18202615_5 . . . 38
3.114 so_9364720_9364799_0 . . . 38
3.115 so_11200805_11200894_1 . . . 38
3.116 so_29346820_29352099_0 . . . 38
3.117 so_22016495_22032011_4 . . . 38
3.118 so_22016495_22032011_7 . . . 38
3.119 so_28118360_28118454_0 . . . 38
3.120 so_37312668_37313013_2 . . . 38
3.121 so_17642624_17648572_1 . . . 38

3.122 so_29997142_29997142_0 . . . 38
3.123 so_23428684_23436665_3 . . . 38
3.124 so_23491781_23655125_2 . . . 38
3.125 so_23491781_23655125_7 . . . 38
3.126 so_23491781_23655125_3 . . . 38
3.127 so_23491781_23655125_1 . . . 38
3.128 so_29346820_29352099_1 . . . 39
3.129 so_29346820_29365435_0 . . . 39
3.130 so_3571959_3571985_0 . . . 39
3.131 so_31565836_31565836_1 . . . 39
3.132 so_23503013_23503013_0 . . . 39
3.133 so_26196095_28446388_2 . . . 39
3.134 so_24018932_24020013_2 . . . 39
3.135 so_32812663_32813409_2 . . . 39
3.136 so_15113437_15113748_0 . . . 39
3.137 so_11336173_11336240_1 . . . 39
3.138 so_22016495_22032011_8 . . . 39
3.139 so_22016495_22032011_5 . . . 39
3.140 so_31381563_31381563_1 . . . 39
3.141 so_31381563_31381563_0 . . . 39
3.142 so_31381563_31381563_4 . . . 39
3.143 so_31381563_31381563_3 . . . 39
3.144 so_29954191_29954631_1 . . . 39
3.145 so_37312668_37313013_1 . . . 39
3.146 so_32031406_32031589_0 . . . 39
3.147 so_42588542_42588542_0 . . . 39
3.148 so_26054494_26054719_2 . . . 39
3.149 so_46207999_46208132_1 . . . 39
3.150 so_28324686_28324791_1 . . . 39
3.151 so_28530583_28534978_3 . . . 39
3.152 so_29067279_29067638_0 . . . 39
3.153 so_34323489_34324971_4 . . . 39
3.154 so_9213985_9214528_2 . . . 39
3.155 so_42582531_42600027_2 . . . 39
3.156 so_42610429_42611583_1 . . . 39
3.157 so_28667924_28701901_0 . . . 39
3.158 so_2671532_2671694_2 . . . 39
3.159 so_45436736_45436736_0 . . . 39
3.160 so_5620008_5620008_1 . . . 39
3.161 so_5620008_5620008_0 . . . 39
3.162 so_4891067_4891102_3 . . . 39
3.163 so_31681788_31683096_0 . . . 39
3.164 so_4639576_4639839_1 . . . 39
3.165 so_33015748_33015952_1 . . . 39
3.166 so_43119643_43120923_1 . . . 39
3.167 so_8091302_8091463_0 . . . 39
3.168 so_28763375_28763570_2 . . . 39
3.169 so_5565522_5565522_0 . . . 39
3.170 so_5565522_5565706_1 . . . 39
3.171 so_5565522_5565706_0 . . . 39
3.172 so_19081578_19082244_0 . . . 39

3.173 so_28839672_28839755_0 . . . 39
3.174 so_28264279_28264640_0 . . . 39
3.175 so_28264279_28462742_2 . . . 39
3.176 so_29997142_29997331_0 . . . 39
3.177 so_26040452_26040651_2 . . . 39
3.178 so_26040452_26040720_0 . . . 39
3.179 so_31383759_31384223_3 . . . 39
3.180 so_12918637_12918985_1 . . . 39
3.181 so_46071992_46072220_1 . . . 39
3.182 so_36965727_36970048_0 . . . 39
3.183 so_33849718_33849802_0 . . . 39
3.184 so_23428684_23436665_1 . . . 39
3.185 so_23428684_23436665_0 . . . 39
3.186 so_23428684_23763205_0 . . . 39
3.187 so_42516175_42516332_0 . . . 39
3.188 so_19630138_19630352_0 . . . 39
3.189 so_33873788_33874942_2 . . . 39
3.190 so_33873788_33874942_1 . . . 39
3.191 so_34445336_34446445_0 . . . 39
3.192 so_5617254_5617254_0 . . . 39
3.193 so_5617254_5617254_1 . . . 39
3.194 so_31843093_32013955_2 . . . 39
3.195 so_28506342_28506460_0 . . . 39
3.196 so_23491781_23655125_5 . . . 39
3.197 so_23491781_23491781_0 . . . 39
3.198 so_28134744_28134991_2 . . . 39
3.199 so_27136524_27136858_0 . . . 39
3.200 so_27136524_27136708_1 . . . 39
3.201 so_30152687_30152687_0 . . . 39
3.202 so_6466816_6467562_0 . . . 39
3.203 so_6466816_6467562_1 . . . 39
3.204 so_42725415_42725415_0 . . . 39
3.205 so_29397864_29404063_0 . . . 39
3.206 so_2128321_2128727_1 . . . 39
3.207 so_47129941_47129941_0 . . . 39
3.208 so_43193749_43194989_0 . . . 39
3.209 so_42747837_42747837_0 . . . 39
3.210 so_42747837_42750838_0 . . . 39
3.211 so_42747837_42750838_1 . . . 39
3.212 so_17139005_17139136_0 . . . 39
3.213 so_19630570_19630570_0 . . . 39
3.214 so_19630570_19630570_1 . . . 39
3.215 so_4518598_4518625_1 . . . 39
3.216 so_28779819_28780125_0 . . . 39
3.217 so_18899102_18899585_1 . . . 39
3.218 so_43467095_43468519_0 . . . 39
3.219 so_43467095_43467280_0 . . . 39
3.220 so_43467095_43467268_1 . . . 39
3.221 so_17677730_17678970_3 . . . 39
3.222 so_26811356_26811447_5 . . . 39
3.223 so_12978745_12978745_0 . . . 39

3.224 so_18028200_18028274_0 . . . 39
3.225 so_15805394_15805532_1 . . . 39
3.226 so_35040614_35041134_0 . . . 39
3.227 so_42120053_42120158_1 . . . 39
3.228 so_14654377_14654428_2 . . . 39
3.229 so_26196095_28446388_0 . . . 39
3.230 so_32812663_32813409_0 . . . 39
3.231 so_10455131_10455390_2 . . . 39
3.232 so_10847176_10847368_5 . . . 39
3.233 so_32031406_32031589_1 . . . 39
3.234 so_14547986_14548934_2 . . . 39
3.235 so_40962505_40962505_0 . . . 39
3.236 so_14372185_14374396_0 . . . 39
3.237 so_17536196_17536450_0 . . . 39
3.238 so_17536196_17536243_3 . . . 39
3.239 so_32141328_32141669_0 . . . 39
3.240 so_28530583_28534978_1 . . . 39
3.241 so_25284863_25285719_0 . . . 39
3.242 so_15865634_15865729_4 . . . 39
3.243 so_45120323_45120376_0 . . . 39
3.244 so_32102486_32102652_1 . . . 39
3.245 so_4891067_4891102_4 . . . 39
3.246 so_42518258_42518371_0 . . . 39
3.247 so_14185847_14185855_0 . . . 39
3.248 so_29287361_29287458_2 . . . 39
3.249 so_14909997_14909999_8 . . . 39
3.250 so_45975114_45975183_0 . . . 39
3.251 so_45900040_45901227_1 . . . 39
3.252 so_8329871_8329930_1 . . . 39
3.253 so_33812659_33812887_1 . . . 39
3.254 so_29347717_29348149_0 . . . 39
3.255 so_36965727_36970048_1 . . . 39
3.256 so_28106661_28106896_0 . . . 39
3.257 so_22581352_22581639_0 . . . 39
3.258 so_908830_910686_0 . . . 39
3.259 so_26241023_26241023_1 . . . 39
3.260 so_36510053_36510397_0 . . . 39
3.261 so_4145915_4146296_1 . . . 39
3.262 so_36841401_36841449_2 . . . 39
3.263 so_38437533_38437714_0 . . . 39
3.264 so_28134744_28134991_1 . . . 39
3.265 so_27136524_27136708_2 . . . 39
3.266 so_27136524_27136858_1 . . . 39
3.267 so_16258601_16258962_0 . . . 39
3.268 so_18629184_18629454_3 . . . 39
3.269 so_18629184_18629404_0 . . . 39
3.270 so_29526585_29526634_1 . . . 39
3.271 so_15960641_15960745_1 . . . 39
3.272 so_15385083_15385263_0 . . . 39
3.273 so_45027922_45028139_1 . . . 39
3.274 so_43193749_43194989_1 . . . 39

3.275 so_34638149_34638478_0 . . . 39
3.276 so_17139005_17139136_1 . . . 39
3.277 so_28763052_28763052_3 . . . 39
3.278 so_9364720_9364799_1 . . . 39
3.279 so_41982545_41983799_0 . . . 39
3.280 so_34835430_34835609_1 . . . 39
3.281 so_18547537_18547537_0 . . . 39
3.282 so_2342550_2342696_3 . . . 39
3.283 so_26811356_26811447_4 . . . 39
3.284 so_9234217_9234299_1 . . . 39
3.285 so_31333301_31333301_0 . . . 39
3.286 so_36508157_36508299_0 . . . 39
3.287 so_30155802_30156450_2 . . . 39
3.288 so_7356603_7357299_1 . . . 39
3.289 so_7356603_7357299_2 . . . 39
3.290 so_6465325_26052100_12 . . . 39
3.291 so_47353796_47354427_0 . . . 39
3.292 so_23494193_23494320_1 . . . 39
3.293 so_5622092_5622092_1 . . . 39
3.294 so_5622092_5622092_0 . . . 39
3.295 so_41966731_41967148_1 . . . 39
3.296 so_37999741_38000200_0 . . . 39
3.297 so_33487729_33487977_0 . . . 40
3.298 so_14389006_14389118_0 . . . 40
3.299 so_42220688_42220956_0 . . . 40
3.300 so_16593204_16593618_1 . . . 40
3.301 so_16593204_16593618_0 . . . 40
3.302 so_1197340_1197490_2 . . . 40
3.303 so_31643595_31644046_0 . . . 40
3.304 so_15891106_15891160_0 . . . 40
3.305 so_24644982_24645738_1 . . . 40
3.306 so_41480854_41481150_3 . . . 40
3.307 so_41480854_41481083_0 . . . 40
3.308 so_41480854_41481150_0 . . . 40
3.309 so_41480854_41481150_1 . . . 40
3.310 so_41480854_41481083_1 . . . 40
3.311 so_43585237_43707540_1 . . . 40
3.312 so_43585237_43707540_0 . . . 40
3.313 so_32504059_32504539_1 . . . 40
3.314 so_32504059_32504539_0 . . . 40
3.315 so_32504059_32504539_2 . . . 40
3.316 so_34149992_34149992_4 . . . 40
3.317 so_34149992_34149992_17 . . . 40
3.318 so_34149992_34149992_9 . . . 40
3.319 so_34149992_34149992_12 . . . 40
3.320 so_33673798_33673798_5 . . . 40
3.321 so_24716801_24716833_1 . . . 40
3.322 so_24716801_24716941_0 . . . 40
3.323 so_11688162_11688435_1 . . . 40
3.324 so_3691835_3692486_2 . . . 40
3.325 so_38597274_38597324_0 . . . 40

3.326 so_36607185_36608602_0 . . . 40
3.327 so_36607185_36607185_1 . . . 40
3.328 so_36607185_36607185_0 . . . 40
3.329 so_34552380_34552544_0 . . . 40
3.330 so_34552380_34552418_0 . . . 40
3.331 so_30264577_30264691_0 . . . 40
3.332 so_41539456_41539584_0 . . . 40
3.333 so_26529842_26530462_3 . . . 40
3.334 so_40962505_40962505_1 . . . 40
3.335 so_18693170_18693170_0 . . . 40
3.336 so_12868576_12868591_0 . . . 40
3.337 so_9305381_9305381_1 . . . 40
3.338 so_28203981_28204207_2 . . . 40
3.339 so_11126496_11126548_0 . . . 40
3.340 so_7670000_7670541_0 . . . 40
3.341 so_7670000_7670544_1 . . . 40
3.342 so_7670000_7670544_0 . . . 40
3.343 so_603390_603400_3 . . . 40
3.344 so_28347008_28347008_0 . . . 40
3.345 so_994353_994428_0 . . . 40
3.346 so_32102486_32102585_0 . . . 40
3.347 so_44245335_44245558_1 . . . 40
3.348 so_44245335_44245558_0 . . . 40
3.349 so_10939543_10940711_0 . . . 40
3.350 so_44308216_44308332_0 . . . 40
3.351 so_28766075_28766204_0 . . . 40
3.352 so_42518258_42518371_4 . . . 40
3.353 so_42518258_42518371_1 . . . 40
3.354 so_42518258_42518371_2 . . . 40
3.355 so_29287361_29287444_2 . . . 40
3.356 so_29287361_29287444_1 . . . 40
3.357 so_33396435_33396622_0 . . . 40
3.358 so_33396435_33396622_2 . . . 40
3.359 so_33396435_33396622_1 . . . 40
3.360 so_27208502_27208585_1 . . . 40
3.361 so_23767241_23767241_0 . . . 40
3.362 so_14909997_14909999_10 . . . 40
3.363 so_44094458_44094815_7 . . . 40
3.364 so_11874494_11874580_3 . . . 40
3.365 so_45900040_45901227_2 . . . 40
3.366 so_39971659_39971691_2 . . . 40
3.367 so_37933470_37933627_1 . . . 40
3.368 so_37933470_37933627_0 . . . 40
3.369 so_41491221_41493715_4 . . . 40
3.370 so_41491221_41493715_2 . . . 40
3.371 so_4494919_4495050_0 . . . 40
3.372 so_46224119_46224187_0 . . . 40
3.373 so_4193639_4193698_0 . . . 40
3.374 so_4193639_4193698_1 . . . 40
3.375 so_12842273_12842471_0 . . . 40
3.376 so_12842273_12842419_2 . . . 40

3.377 so_12842273_12842419_1 . . . 40
3.378 so_12842273_12842419_0 . . . 40
3.379 so_6688118_6688626_2 . . . 40
3.380 so_6688118_6688626_4 . . . 40
3.381 so_5337370_5337391_4 . . . 40
3.382 so_32353894_32354036_0 . . . 40
3.383 so_29347717_29348149_1 . . . 40
3.384 so_18860895_42069703_0 . . . 40
3.385 so_35398013_35398101_2 . . . 40
3.386 so_35398013_35398101_0 . . . 40
3.387 so_28916602_28916778_1 . . . 40
3.388 so_10211273_10211420_0 . . . 40
3.389 so_16760983_16761002_1 . . . 40
3.390 so_16760983_16761002_0 . . . 40
3.391 so_22581352_22581639_1 . . . 40
3.392 so_908830_910686_4 . . . 40
3.393 so_4143946_4144086_1 . . . 40
3.394 so_4143946_4144086_0 . . . 40
3.395 so_4143946_4144086_2 . . . 40
3.396 so_34717823_34721371_6 . . . 40
3.397 so_34717823_34721371_5 . . . 40
3.398 so_34717823_34721371_4 . . . 40
3.399 so_26241023_26241098_0 . . . 40
3.400 so_26241023_26241098_2 . . . 40
3.401 so_14951200_14953545_0 . . . 40
3.402 so_21771901_21772170_4 . . . 40
3.403 so_21771901_21772170_2 . . . 40
3.404 so_21771901_21772170_0 . . . 40
3.405 so_21771901_21772170_1 . . . 40
3.406 so_32879038_32882323_1 . . . 40
3.407 so_34986335_34986350_4 . . . 40
3.408 so_4145915_4146034_0 . . . 40
3.409 so_36841401_36841449_4 . . . 40
3.410 so_36841401_36841449_3 . . . 40
3.411 so_27394577_27394577_0 . . . 40
3.412 so_27394577_27394577_3 . . . 40
3.413 so_27394577_27394577_1 . . . 40
3.414 so_7562096_7562583_0 . . . 40
3.415 so_19366615_19367047_0 . . . 40
3.416 so_19366615_19367047_1 . . . 40
3.417 so_19366615_19367075_0 . . . 40
3.418 so_19366615_19367047_2 . . . 40
3.419 so_18629184_18629454_5 . . . 40
3.420 so_18629184_18629454_7 . . . 40
3.421 so_18629184_18629454_4 . . . 40
3.422 so_18629184_18629404_1 . . . 40
3.423 so_15689793_15689925_4 . . . 40
3.424 so_36411023_36411363_0 . . . 40
3.425 so_43736349_43736576_1 . . . 40
3.426 so_15960641_15960745_2 . . . 40
3.427 so_32405505_32405618_2 . . . 40

3.428 so_46450437_46450695_1 . . . 40
3.429 so_46450437_46450695_0 . . . 40
3.430 so_23150761_23150939_1 . . . 40
3.431 so_1144264_1144404_1 . . . 40
3.432 so_23494193_23494320_2 . . . 40
3.433 so_41583524_41584059_0 . . . 40
3.434 so_17139005_17139005_0 . . . 40
3.435 so_17139005_17139136_2 . . . 40
3.436 so_17139005_17139163_0 . . . 40
3.437 so_9364720_9364985_2 . . . 40
3.438 so_9364720_9364799_2 . . . 40
3.439 so_9364720_9364985_3 . . . 40
3.440 so_9364720_9364799_3 . . . 40
3.441 so_9364720_9364985_4 . . . 40
3.442 so_9364720_9364985_0 . . . 40
3.443 so_9364720_9364985_1 . . . 40
3.444 so_9364720_9364985_6 . . . 40
3.445 so_9364720_9364985_5 . . . 40
3.446 so_607259_33733088_0 . . . 40
3.447 so_17667098_17667265_0 . . . 40
3.448 so_29904114_29904347_5 . . . 40
3.449 so_7719814_7719884_0 . . . 40
3.450 so_37604778_37604880_0 . . . 40
3.451 so_14403796_14404244_2 . . . 40
3.452 so_18544249_18545856_3 . . . 40
3.453 so_37901693_37901693_0 . . . 40
3.454 so_37901693_37901693_1 . . . 40
3.455 so_2342550_2342696_4 . . . 40
3.456 so_5622092_5622270_3 . . . 40
3.457 so_5622092_5622270_4 . . . 40
3.458 so_1284529_1284547_0 . . . 40
3.459 so_12214112_12214231_0 . . . 40
3.460 so_17639663_17640116_1 . . . 40
3.461 so_17639663_17640116_0 . . . 40
3.462 so_17639663_17640116_4 . . . 40
3.463 so_17639663_17640116_2 . . . 40
3.464 Scope . . . 41
3.465 so_24293311_24313401_1 . . . 41
3.466 so_24293311_24313401_0 . . . 41
3.467 so_34247975_34248633_3 . . . 41
3.468 so_23684431_23684431_0 . . . 41
3.469 so_34586141_34595754_0 . . . 41
3.470 so_33856187_33856407_0 . . . 41
3.471 so_4151203_4151620_0 . . . 41
3.472 so_27402292_27404546_10 . . . 41
3.473 so_16362551_16362649_0 . . . 41
3.474 so_23966921_23967971_7 . . . 41
3.475 so_23966921_23967971_3 . . . 41
3.476 so_23966921_23967971_5 . . . 41
3.477 so_23966921_23967971_6 . . . 41
3.478 so_23966921_23967971_4 . . . 41

3.479 so_23966921_23967971_2 . . . 41
3.480 so_23966921_23966921_1 . . . 41
3.481 so_31708705_31710965_1 . . . 41
3.482 so_15580903_15580915_0 . . . 41
3.483 so_5987213_5988726_0 . . . 41
3.484 so_24676876_24677389_0 . . . 41
3.485 so_29775091_29775091_1 . . . 41
3.486 so_29775091_29776086_0 . . . 41
3.487 so_29775091_29775973_5 . . . 41
3.488 so_17376399_17376421_1 . . . 41
3.489 so_17376399_17376421_0 . . . 41
3.490 so_23684431_23684431_1 . . . 42
3.491 so_23684431_23684736_0 . . . 42
3.492 so_31759210_31759360_0 . . . 42
3.493 so_16081289_16081662_0 . . . 42
3.494 so_29775091_29775091_2 . . . 42
3.495 so_29775091_29775238_0 . . . 42
3.496 so_24293311_24313401_2 . . . 42
3.497 so_23367912_23368705_0 . . . 42
3.498 so_31708705_31710965_2 . . . 42
3.499 so_16642027_16642148_1 . . . 42
3.500 so_27895689_27895689_0 . . . 42
3.501 so_24293311_24313401_8 . . . 42
3.502 so_23367912_23367912_0 . . . 42
3.503 so_23367912_23367912_1 . . . 42
3.504 so_23367912_23368809_1 . . . 42
3.505 so_34247975_34248633_5 . . . 42
3.506 so_16772759_16773096_1 . . . 42
3.507 so_23709019_23720793_1 . . . 42
3.508 so_23709019_23709019_1 . . . 42
3.509 so_23709019_23720793_2 . . . 42
3.510 so_23709019_23709019_0 . . . 42
3.511 so_23709019_23720793_0 . . . 42
3.512 so_42343155_42344463_4 . . . 42
3.513 so_26341979_26342143_0 . . . 42
3.514 so_31224886_31225016_3 . . . 42
3.515 so_31224886_31225016_4 . . . 42
3.516 so_31224886_31225016_5 . . . 42
3.517 so_30155802_30156450_6 . . . 42
3.518 so_23654246_23654486_2 . . . 42
3.519 so_28135075_28135194_0 . . . 42
3.520 so_28135075_28135194_2 . . . 42
3.521 so_31220154_31220649_0 . . . 42
3.522 so_31220154_31220649_2 . . . 42
3.523 so_31220154_31220649_1 . . . 42
3.524 so_28752247_28768636_1 . . . 42
3.525 so_28752247_28768636_0 . . . 42
3.526 so_26067473_26069040_0 . . . 42
3.527 so_23721731_23721731_0 . . . 42
3.528 so_29775091_29776086_1 . . . 42
3.529 so_29775091_29775973_6 . . . 42

3.530 so_23973281_23973320_0 . . . 42
3.531 so_31713682_31713682_0 . . . 42
3.532 so_37037842_37038369_1 . . . 42
3.533 so_37037842_37038369_0 . . . 42
3.534 so_17376399_17376399_0 . . . 42
3.535 so_23831077_23831077_1 . . . 42
3.536 so_23831077_23831761_1 . . . 42
3.537 so_15746271_15746272_0 . . . 42
3.538 so_29995642_30002714_0 . . . 42
3.539 so_8522055_8522133_0 . . . 42
3.540 so_44294121_44295205_0 . . . 42
3.541 so_44294121_44295205_1 . . . 42
3.542 so_13750653_13750653_0 . . . 42
3.543 so_5474349_5475143_1 . . . 42
3.544 so_5474349_5475143_0 . . . 42
3.545 so_5474349_5474446_0 . . . 42
3.546 so_34247975_34248633_16 . . . 42
3.547 so_34247975_34248633_14 . . . 42
3.548 so_34247975_34248633_15 . . . 42
3.549 so_34247975_34247975_1 . . . 42
3.550 so_7518412_7518775_0 . . . 42
3.551 so_6189096_6189124_1 . . . 42
3.552 so_6189096_6189122_1 . . . 42
3.553 so_6189096_6189122_0 . . . 42
3.554 so_10359289_10359339_2 . . . 42
3.555 so_10359289_10359339_1 . . . 42
3.556 so_7378492_7378547_0 . . . 42
3.557 so_13300285_13300655_0 . . . 42
3.558 so_13300285_13300655_1 . . . 42
3.559 so_29525797_29525852_0 . . . 42
3.560 so_29525797_29525932_3 . . . 42
3.561 so_29525797_29525932_1 . . . 42
3.562 so_29525797_29525852_1 . . . 42
3.563 so_29392234_29392868_0 . . . 42
3.564 so_11596898_11597196_0 . . . 42
3.565 so_9820027_9820580_0 . . . 42
3.566 so_9820027_9820155_1 . . . 42
3.567 so_9820027_9820155_0 . . . 42
3.568 so_3173462_3173477_1 . . . 42
3.569 so_3173462_3173477_0 . . . 42
3.570 so_8489215_8489499_0 . . . 42
3.571 so_6841130_6970226_2 . . . 42
3.572 so_4114794_7378683_1 . . . 42
3.573 so_4114794_7378683_0 . . . 42
3.574 so_10723892_10724143_1 . . . 42
3.575 so_10723892_10724143_2 . . . 42
3.576 so_10723892_10734635_4 . . . 42
3.577 so_8595061_8595523_0 . . . 42
3.578 so_26793979_26829733_0 . . . 42
3.579 so_37409590_37411089_1 . . . 42
3.580 so_17938743_17939275_1 . . . 42

3.581 so_5374319_5374363_0 . . . 42
3.582 so_23991085_23991312_0 . . . 42
3.583 so_4343571_4343626_1 . . . 42
3.584 so_4343571_4343626_0 . . . 42
3.585 so_4343571_4343601_0 . . . 42
3.586 so_36884905_36885464_1 . . . 42
3.587 so_36884905_36885464_0 . . . 42
3.588 so_36922180_36922424_0 . . . 42
3.589 so_23415661_23415661_0 . . . 42
3.590 so_23415661_23415661_1 . . . 42
3.591 so_36304511_36304579_0 . . . 42
3.592 so_36304511_36304579_1 . . . 42
3.593 so_29130530_29130566_1 . . . 42
3.594 so_29130530_29130566_0 . . . 42
3.595 so_16594519_16594619_1 . . . 42
3.596 so_16594519_16594619_0 . . . 42
3.597 so_11963136_11964942_0 . . . 42
3.598 so_30103639_30103878_0 . . . 42
3.599 so_30873671_30874396_0 . . . 42
3.600 so_17607888_17607935_0 . . . 42
3.601 so_17607888_17607935_1 . . . 42
3.602 so_23772928_23772980_0 . . . 42
3.603 so_42119710_42119940_1 . . . 42
3.604 so_42119710_42119940_2 . . . 42
3.605 so_9467868_9468380_0 . . . 42
3.606 so_9467868_9468380_1 . . . 42
3.607 so_25576495_25576588_2 . . . 42
3.608 so_25576495_25576588_1 . . . 42
3.609 so_17658326_17659594_2 . . . 42
3.610 so_40513156_40513319_0 . . . 42
3.611 so_42743884_42743972_3 . . . 42
3.612 so_42743884_42743972_1 . . . 42
3.613 so_31033746_31038983_1 . . . 42
3.614 so_12791704_12791732_1 . . . 42
3.615 so_12791704_12791732_0 . . . 42
3.616 so_26758267_26758357_0 . . . 42
3.617 so_26758267_26758357_2 . . . 42
3.618 so_26758267_26758393_0 . . . 42
3.619 so_26758267_26758393_1 . . . 42
3.620 so_22098834_22099297_2 . . . 42
3.621 so_22098834_22099297_0 . . . 42
3.622 so_26658776_26658776_1 . . . 42
3.623 so_26658776_26658776_0 . . . 42
3.624 so_26274207_26274688_0 . . . 42
3.625 so_13750653_13750653_1 . . . 42
3.626 so_10723892_10734635_5 . . . 42
3.627 so_27651183_27651369_1 . . . 42
3.628 so_27651183_27651369_0 . . . 42
3.629 so_4343571_4343626_3 . . . 42
3.630 so_4343571_4343626_2 . . . 42
3.631 so_29130530_29130566_3 . . . 42

3.632 so_29130530_29130566_2 . . . 42
3.633 so_42743884_42743972_4 . . . 42
3.634 so_42743884_42743972_5 . . . 42
3.635 so_39059511_39086337_2 . . . 43
3.636 so_35695071_35695071_0 . . . 43
3.637 so_25864627_25864755_1 . . . 43
3.638 so_33335752_33335787_1 . . . 43
3.639 so_33335752_33335787_2 . . . 43
3.640 so_25864627_25864690_1 . . . 43
3.641 so_24299963_24303296_2 . . . 43
3.642 so_29392234_29392868_1 . . . 43
3.643 so_37115503_37115963_0 . . . 43
3.644 so_24095071_24290421_1 . . . 43
3.645 so_24095071_24290421_2 . . . 43
3.646 so_10924047_10924198_0 . . . 43
3.647 so_37742749_37743365_0 . . . 43
3.648 so_31033746_31038983_0 . . . 43
3.649 so_26274207_26274688_1 . . . 43
3.650 so_31696292_31696559_0 . . . 43
3.651 so_26168445_26168445_3 . . . 43
3.652 so_26168445_26168445_6 . . . 43
3.653 so_26168445_26168445_0 . . . 43
3.654 so_27890699_27890824_1 . . . 43
3.655 so_27890699_27890824_4 . . . 43
3.656 so_29827850_29828989_1 . . . 43
3.657 so_29827850_29828989_3 . . . 43
3.658 so_40257011_40257321_5 . . . 43
3.659 so_31875710_31876176_1 . . . 43
3.660 so_31875710_31876176_4 . . . 43
3.661 so_31875710_31876176_7 . . . 43
3.662 so_27415711_27415938_3 . . . 43
3.663 so_27415711_27415938_0 . . . 43
3.664 so_27415711_27415938_6 . . . 43
3.665 so_24423481_24424033_0 . . . 43
3.666 so_29198354_29198354_3 . . . 43
3.667 so_29198354_29198354_6 . . . 43
3.668 so_29198354_29198354_0 . . . 43
3.669 so_28060382_28060940_2 . . . 43
3.670 so_28060382_28060382_2 . . . 43
3.671 so_8070233_8070789_0 . . . 43
3.672 so_28241874_28242034_3 . . . 43
3.673 so_28241874_28242034_6 . . . 43
3.674 so_28241874_28242038_2 . . . 43
3.675 so_37219984_39203657_5 . . . 43
3.676 so_37219984_39203657_11 . . . 43
3.677 so_37219984_39203657_8 . . . 43
3.678 so_13138605_13139474_0 . . . 43
3.679 so_13138605_13139474_2 . . . 43
3.680 so_13138605_13139474_1 . . . 43
3.681 so_41471439_41472929_4 . . . 43
3.682 so_37744047_37744321_2 . . . 43

3.683 so_37744047_37744321_5 . . . 43
3.684 so_32037146_32037250_1 . . . 43
3.685 so_5149119_5149468_5 . . . 43
3.686 so_31769853_31769853_1 . . . 43
3.687 so_23367912_23367912_2 . . . 44
3.688 so_23367912_23368705_1 . . . 44
3.689 so_34559689_34562045_0 . . . 44
3.690 so_3898757_3898787_3 . . . 44
3.691 so_35654908_35655727_0 . . . 44
3.692 so_24459043_24459211_1 . . . 44
3.693 so_36224099_36224537_0 . . . 44
3.694 so_23721731_23721731_1 . . . 44
3.695 so_8522055_8522133_1 . . . 44
3.696 so_28288110_28288167_5 . . . 44
3.697 so_29525797_29525932_0 . . . 44
3.698 so_18880611_18880713_1 . . . 44
3.699 so_16969535_16970198_0 . . . 44
3.700 so_4075276_4075278_0 . . . 44
3.701 so_4075276_4075307_0 . . . 44
3.702 so_37541749_37542104_0 . . . 44
3.703 so_23725571_23725687_3 . . . 44
3.704 so_27460527_27462301_0 . . . 44
3.705 so_17658326_17659594_1 . . . 44
3.706 so_19344955_19345096_3 . . . 44
3.707 so_33462958_33463344_1 . . . 44
3.708 so_35724423_35725114_0 . . . 44
3.709 so_35724423_35725486_0 . . . 44
3.710 so_30125671_30126032_0 . . . 44
3.711 so_24839157_24841967_1 . . . 44
3.712 so_5677097_5677097_1 . . . 44
3.713 so_5677097_5677097_0 . . . 44
3.714 so_36295738_37138998_8 . . . 44
3.715 so_37541749_37542104_2 . . . 44
3.716 so_23725571_23725687_4 . . . 44
3.717 so_23731821_23731821_0 . . . 44
3.718 so_26758267_26758393_2 . . . 44
3.719 so_30125671_30126032_1 . . . 44
3.720 so_28019026_28019026_0 . . . 44
3.721 so_35846080_35846423_1 . . . 44
3.722 so_30125671_30126032_2 . . . 44
3.723 so_12351460_34539306_0 . . . 44
3.724 so_16969535_16970198_1 . . . 44
3.725 so_41796441_41796523_1 . . . 44
3.726 so_41796441_41796523_0 . . . 44
3.727 so_41523989_41524059_0 . . . 44
3.728 so_34497208_34497208_0 . . . 44
3.729 so_20188241_20189131_2 . . . 44
3.730 so_24393273_24393273_0 . . . 44
3.731 so_23725995_23725995_0 . . . 44
3.732 so_23725571_23725687_1 . . . 44
3.733 so_23725571_23725633_0 . . . 44

3.734 so_28060382_28060382_3 . . . 44
3.735 so_28060247_28060247_0 . . . 44
3.736 so_22041376_22041425_0 . . . 45
3.737 so_34247975_34247975_0 . . . 45
3.738 so_34247975_34249649_3 . . . 45
3.739 so_34247975_34248633_1 . . . 45
3.740 so_1368642_1368738_0 . . . 45
3.741 so_28163389_28163505_0 . . . 45
3.742 so_6841130_6970226_0 . . . 45
3.743 so_24409996_24409996_0 . . . 45
3.744 so_17938743_17939275_0 . . . 45
3.745 so_23966921_23966921_2 . . . 45
3.746 so_23966921_23967971_0 . . . 45
3.747 so_23966921_23966921_0 . . . 45
3.748 so_31729985_31731551_5 . . . 45
3.749 so_37301946_37303140_3 . . . 45
3.750 so_31494906_31495054_0 . . . 45
3.751 so_19541086_19541301_0 . . . 45
3.752 so_9622874_9623023_0 . . . 45
3.753 so_28103318_28103433_0 . . . 45
3.754 so_28103318_28103318_0 . . . 45
3.755 so_3279781_3309137_1 . . . 45
3.756 so_1368642_1368738_1 . . . 45
3.757 so_3279781_3309137_2 . . . 45
3.758 so_6841130_6970226_1 . . . 45
3.759 so_14794547_14794559_1 . . . 45
3.760 so_21286497_21286621_0 . . . 45
3.761 so_33569992_33570258_0 . . . 46
3.762 so_34449966_34451597_1 . . . 46
3.763 so_35872045_35872378_0 . . . 46
3.764 so_23947055_23947470_1 . . . 46
3.765 so_23947055_23947470_0 . . . 46
3.766 so_23947055_23947470_2 . . . 46
3.767 so_10062156_10062510_2 . . . 46
3.768 so_10062156_10062510_0 . . . 46
3.769 so_12943607_12943770_0 . . . 46
3.770 so_42506850_42507317_0 . . . 46
3.771 so_28029722_28030481_2 . . . 46
3.772 so_28029722_28030481_0 . . . 46
3.773 so_13094898_13095199_0 . . . 46
3.774 so_11825902_11827016_1 . . . 46
3.775 so_11825902_11827016_2 . . . 46
3.776 so_24299963_24303296_0 . . . 46
3.777 so_28185147_28189804_0 . . . 46
3.778 so_28185147_28189804_1 . . . 46
3.779 so_28185147_28185147_0 . . . 46
3.780 so_27316607_27316980_1 . . . 46
3.781 so_18160290_18160890_0 . . . 46
3.782 so_18160290_18160890_1 . . . 46
3.783 so_7040076_7041294_0 . . . 46
3.784 so_7040076_7041294_1 . . . 46

3.785 so_28096664_28096988_2 . . . 46
3.786 so_28096664_28096988_0 . . . 46
3.787 so_29196292_29197173_0 . . . 46
3.788 so_29196292_29197134_0 . . . 46
3.789 so_3601792_3602671_1 . . . 46
3.790 so_3601792_3602671_0 . . . 46
3.791 so_16452922_16452922_0 . . . 46
3.792 so_33267529_33267641_1 . . . 46
3.793 so_33267529_33267641_3 . . . 46
3.794 so_26420320_26421623_0 . . . 46
3.795 so_19231959_19232203_0 . . . 46
3.796 so_44913311_44913311_0 . . . 46
3.797 so_31484414_31484414_0 . . . 46
3.798 so_26681873_26682543_0 . . . 46
3.799 so_26681873_26682543_1 . . . 46
3.800 so_41515989_41516179_0 . . . 46
3.801 so_24095412_24095412_2 . . . 46
3.802 so_34532892_34532892_0 . . . 46
3.803 so_7357078_7357103_0 . . . 46
3.804 so_24038003_24038248_0 . . . 46
3.805 so_24038003_24038003_1 . . . 46
3.806 so_31720516_31723214_0 . . . 46
3.807 so_24168137_24168426_2 . . . 46
3.808 so_32707990_32708142_0 . . . 46
3.809 so_23758306_23758306_0 . . . 46
3.810 so_23750766_23751003_0 . . . 46
3.811 so_23750766_23751003_1 . . . 46
3.812 so_4274386_4279182_2 . . . 46
3.813 so_12187549_12187709_0 . . . 46
3.814 so_7898857_7898947_0 . . . 46
3.815 so_7898857_7898857_0 . . . 46
3.816 so_25031253_25031961_7 . . . 46
3.817 so_25031253_25031961_6 . . . 46
3.818 so_25031253_25031961_5 . . . 46
3.819 so_25031253_25031961_4 . . . 46
3.820 so_26761385_26761385_0 . . . 46
3.821 so_26761385_26761385_1 . . . 46
3.822 so_30047021_30048451_1 . . . 46
3.823 so_45384956_45385086_0 . . . 46
3.824 so_4884494_4885497_0 . . . 46
3.825 so_4884494_4885497_1 . . . 46
3.826 so_25512673_25512970_1 . . . 46
3.827 so_25512673_25512970_2 . . . 46
3.828 so_35926728_35927869_0 . . . 46
3.829 so_19564009_19566059_3 . . . 46
3.830 so_10924047_10924198_1 . . . 46
3.831 so_10924047_10924193_1 . . . 46
3.832 so_23773504_23773504_2 . . . 46
3.833 so_23773504_23773504_1 . . . 46
3.834 so_23773504_23773504_0 . . . 46
3.835 so_29860124_29860677_0 . . . 46

3.836 so_28932026_28932420_5 . . . 46
3.837 so_15537023_15538759_0 . . . 46
3.838 so_21461315_21461748_0 . . . 46
3.839 so_32220551_32220880_2 . . . 46
3.840 so_32220551_32220880_0 . . . 46
3.841 so_20215531_20218433_0 . . . 46
3.842 so_22784926_22785370_2 . . . 46
3.843 so_13444626_13444894_0 . . . 46
3.844 so_282800_26695750_2 . . . 46
3.845 so_26274207_26274688_3 . . . 46
3.846 so_26274207_26274688_2 . . . 46
3.847 so_22792523_22792523_0 . . . 46
3.848 so_34614869_34615030_2 . . . 46
3.849 so_34614869_34615030_3 . . . 46
3.850 so_34614869_34615030_1 . . . 46
3.851 so_34614869_34615030_4 . . . 46
3.852 so_34614869_34615030_0 . . . 46
3.853 so_24293311_24313401_5 . . . 46
3.854 so_31655054_31655330_1 . . . 46
3.855 so_31655054_31655330_0 . . . 46
3.856 so_31655054_31655330_2 . . . 46
3.857 so_31655054_31655330_4 . . . 46
3.858 so_31655054_31655330_3 . . . 46
3.859 so_23712025_23712633_1 . . . 46
3.860 so_4294857_4294885_0 . . . 46
3.861 so_30047021_30048451_6 . . . 46
3.862 so_30047021_30048451_5 . . . 46
3.863 so_30047021_30048451_4 . . . 46
3.864 so_30047021_30048451_2 . . . 46
3.865 so_30047021_30048451_3 . . . 46
3.866 so_23797487_23797487_0 . . . 47
3.867 so_28526061_28526122_0 . . . 47
3.868 so_28992265_28992608_0 . . . 47
3.869 so_28241874_28241874_0 . . . 47
3.870 so_5987213_5987276_0 . . . 47
3.871 so_5987213_5987276_1 . . . 47
3.872 so_5987213_5987276_2 . . . 47
3.873 so_5988621_5988621_0 . . . 47
3.874 so_5988621_5988746_0 . . . 47
3.875 so_5987213_5987213_0 . . . 48
3.876 so_27433544_27434098_0 . . . 48
3.877 so_45652203_45652215_0 . . . 48
3.878 so_41670800_41670857_1 . . . 48
3.879 so_26431190_26431541_0 . . . 48
3.880 so_5406533_5911476_0 . . . 48
3.881 so_8489215_8489499_1 . . . 48
3.882 so_12143836_12144590_1 . . . 48
3.883 so_5187891_5187935_0 . . . 48
3.884 so_33717196_33717247_0 . . . 48
3.885 so_20814505_20814545_0 . . . 48
3.886 so_4438281_4438384_0 . . . 48

3.887 so_13898053_13898517_0 . . . 48
3.888 so_23582526_23582874_0 . . . 48
3.889 so_23582526_23583080_1 . . . 48
3.890 so_27460527_27462301_1 . . . 48
3.891 so_24129551_24129588_1 . . . 48
3.892 so_11359562_11359733_0 . . . 48
3.893 so_17658326_17659594_0 . . . 48
3.894 so_31702956_31703369_0 . . . 48
3.895 so_31702956_31702956_0 . . . 48
3.896 so_23329382_23341289_0 . . . 48
3.897 so_31711183_31711699_1 . . . 48
3.898 so_31711183_31711422_2 . . . 48
3.899 so_4094008_4094068_0 . . . 48
3.900 so_5765924_5765924_0 . . . 48
3.901 so_23709019_23720793_3 . . . 48
3.902 so_5401575_5401792_1 . . . 48
3.903 so_26341979_26342136_0 . . . 48
3.904 so_34839886_34839955_0 . . . 48
3.905 so_11363737_11364760_0 . . . 48
3.906 so_16592988_16593453_0 . . . 48
3.907 so_4163799_4163806_5 . . . 48
3.908 so_29176179_29176486_0 . . . 48
3.909 so_38393792_38393880_0 . . . 48
3.910 so_21401944_21402811_0 . . . 48
3.911 so_38111837_38111950_0 . . . 48
3.912 so_28532535_28532616_0 . . . 48
3.913 so_282800_26695750_1 . . . 48
3.914 so_19250608_19250664_0 . . . 48
3.915 so_19250608_19250829_0 . . . 48
3.916 so_31702956_31702956_1 . . . 48
3.917 so_31702956_31703369_1 . . . 48
3.918 so_34158902_34159684_0 . . . 48
3.919 so_12568776_12568847_1 . . . 48
3.920 so_34061864_34061982_0 . . . 48
3.921 so_24160600_24160661_0 . . . 48
3.922 so_40489677_40490475_1 . . . 48
3.923 Name lookup . . . 48
3.924 so_4448598_4448686_0 . . . 48
3.925 so_30232485_30233005_0 . . . 48
3.926 so_24016605_24018720_1 . . . 48
3.927 so_9303843_9306682_2 . . . 48
3.928 so_5195512_5195657_0 . . . 48
3.929 so_23976789_23976808_0 . . . 48
3.930 so_27679567_27679640_0 . . . 48
3.931 so_34154770_34154770_1 . . . 48
3.932 so_3368129_3368191_0 . . . 48
3.933 so_18402753_18403086_1 . . . 48
3.934 so_44607634_45719873_0 . . . 48
3.935 so_44607634_44607634_0 . . . 48
3.936 so_30245091_30245730_1 . . . 48
3.937 so_3617866_3617866_0 . . . 48

3.938 so_26022439_26022581_0 . . . 48
3.939 so_15342410_15342557_2 . . . 48
3.940 so_23907478_23907478_0 . . . 48
3.941 so_23966473_23966473_0 . . . 48
3.942 so_28932026_28932231_1 . . . 48
3.943 so_31702956_31703369_2 . . . 48
3.944 so_6356994_6356994_0 . . . 48
3.945 so_24470404_24470404_0 . . . 48
3.946 so_7025054_7025054_0 . . . 49
3.947 so_5149119_5149468_6 . . . 49
3.948 so_38572374_38575475_1 . . . 49
3.949 so_15342410_15342557_1 . . . 49
3.950 so_40577179_40577244_5 . . . 49
3.951 so_14492878_14493136_0 . . . 49
3.952 so_33746865_33747605_1 . . . 49
3.953 so_46992006_46993212_0 . . . 49
3.954 so_4135312_4135545_1 . . . 49
3.955 so_4448598_4448686_1 . . . 49
3.956 so_43890123_43890241_0 . . . 49
3.957 so_21005172_21005338_1 . . . 49
3.958 so_26850784_26851159_0 . . . 49
3.959 so_17826307_17826307_0 . . . 49
3.960 so_25863485_25865297_0 . . . 49
3.961 so_23976789_23976789_0 . . . 49
3.962 so_27316607_27316852_1 . . . 49
3.963 so_25887448_25887601_0 . . . 49
3.964 so_7444790_7444919_0 . . . 49
3.965 so_31779642_31779642_0 . . . 49
3.966 so_27402292_27404546_6 . . . 49
3.967 so_27402292_27404825_0 . . . 49
3.968 so_24533091_24533452_3 . . . 49
3.969 so_23001688_23001740_0 . . . 49
3.970 so_16696989_16698562_1 . . . 49
3.971 so_7210496_7210512_1 . . . 49
3.972 so_23966473_23966473_1 . . . 49
3.973 so_34278868_37040939_2 . . . 49
3.974 so_21384604_21385079_3 . . . 49
3.975 so_23966271_23966271_0 . . . 49
3.976 so_17437073_17437594_1 . . . 49
3.977 so_23577711_23577711_0 . . . 49
3.978 so_7444790_7444919_1 . . . 49
3.979 so_26070916_26071134_2 . . . 49
3.980 so_31490710_31490957_0 . . . 49
3.981 so_37301946_37303140_5 . . . 49
3.982 so_18933857_18934399_0 . . . 49
3.983 so_4396356_4396356_0 . . . 49
3.984 so_12170562_12170694_1 . . . 49
3.985 so_23822789_23822789_0 . . . 49
3.986 so_23958590_23958903_0 . . . 49
3.987 so_23958590_23958903_1 . . . 49
3.988 so_35373384_35375162_10 . . . 49

3.989 so_31772588_31775526_0 . . . 49
3.990 so_31772588_31773124_0 . . . 49
3.991 so_23831077_23831077_0 . . . 49
3.992 so_23831077_23831761_0 . . . 49
3.993 so_7965749_7965749_0 . . . 49
3.994 so_19728109_19728661_0 . . . 49
3.995 so_25031253_25031961_0 . . . 49
3.996 so_37301946_37303140_4 . . . 49
3.997 so_23849100_23849543_0 . . . 49
3.998 so_25031253_25031961_1 . . . 49
3.999 so_26850784_26851159_1 . . . 49
3.1000 so_36201870_36202451_0 . . . 49
3.1001 so_5195512_5195657_1 . . . 49
3.1002 so_24228299_24228299_1 . . . 49
3.1003 so_18475953_18489952_0 . . . 49
3.1004 so_9879719_9880459_0 . . . 49
3.1005 so_23060007_23060119_0 . . . 49
3.1006 so_27402292_27404546_1 . . . 49
3.1007 so_29019462_29019462_0 . . . 49
3.1008 so_42076314_42076581_0 . . . 49
3.1009 so_34278868_37040939_3 . . . 49
3.1010 so_17437073_17437594_2 . . . 49
3.1011 so_10866895_10867345_1 . . . 49
3.1012 so_23849100_23849543_1 . . . 49
3.1013 so_16493015_16498399_1 . . . 50
3.1014 so_23947055_23947470_3 . . . 50
3.1015 so_27583384_27583384_0 . . . 50
3.1016 so_24293311_24293311_0 . . . 50
3.1017 so_24293311_24293443_0 . . . 50
3.1018 so_4296924_4297052_5 . . . 50
3.1019 so_4296924_4297052_6 . . . 50
3.1020 so_4296924_4297052_4 . . . 50
3.1021 so_4296924_4297052_1 . . . 50
3.1022 so_27316607_27316852_2 . . . 50
3.1023 so_18475953_18489952_4 . . . 50
3.1024 so_18475953_18489952_3 . . . 50
3.1025 so_3608305_3608326_3 . . . 50
3.1026 so_3608305_3608326_1 . . . 50
3.1027 so_3608305_3608326_4 . . . 50
3.1028 so_3608305_3608326_0 . . . 50
3.1029 so_3608305_3608326_2 . . . 50
3.1030 so_5149119_5149468_4 . . . 50
3.1031 so_5149119_5149468_1 . . . 50
3.1032 so_5149119_5149468_3 . . . 50
3.1033 so_5149119_5149468_2 . . . 50
3.1034 so_16766137_16766850_0 . . . 50
3.1035 so_27402292_27404825_1 . . . 50
3.1036 so_27402292_27404546_4 . . . 50
3.1037 so_851584_851760_1 . . . 50
3.1038 so_31202899_31203015_0 . . . 50
3.1039 so_36545819_36548889_1 . . . 50

3.1040 so_25031253_25031961_2 . . . 50
3.1041 so_40577179_40577244_6 . . . 50
3.1042 so_23906439_23906439_0 . . . 50
3.1043 so_7210496_7210512_3 . . . 50
3.1044 so_33907864_33908092_3 . . . 50
3.1045 so_33907864_33908092_5 . . . 50
3.1046 so_33907864_33908092_4 . . . 50
3.1047 so_33907864_33908092_0 . . . 50
3.1048 so_16883333_16883333_0 . . . 50
3.1049 so_16666304_16666387_1 . . . 50
3.1050 so_16666304_16666387_2 . . . 50
3.1051 so_27371102_27371352_0 . . . 50
3.1052 so_46992006_46993212_2 . . . 51
3.1053 so_46992006_46993212_1 . . . 51
3.1054 so_31457627_31457828_0 . . . 51
3.1055 so_31457627_31457627_0 . . . 51
3.1056 so_31457627_31457627_1 . . . 51
3.1057 so_35286778_35286778_0 . . . 51
3.1058 so_4296924_4297052_2 . . . 51
3.1059 so_25863485_25865297_1 . . . 51
3.1060 so_18475953_18489952_1 . . . 51
3.1061 so_18475953_18489952_5 . . . 51
3.1062 so_25887448_25887601_1 . . . 51
3.1063 so_27402292_27404546_5 . . . 51
3.1064 so_27402292_27404825_3 . . . 51
3.1065 so_27402292_27404546_11 . . . 51
3.1066 so_27402292_27404546_2 . . . 51
3.1067 so_31393043_31393043_1 . . . 51
3.1068 so_31393043_31393043_0 . . . 51
3.1069 so_29574616_29575182_0 . . . 51
3.1070 so_33907864_33908092_1 . . . 51
3.1071 so_30941198_30941308_0 . . . 51
3.1072 so_40674779_40685194_0 . . . 51
3.1073 so_40674779_40685194_1 . . . 51
3.1074 so_40674779_40685194_4 . . . 51
3.1075 so_20808993_20809467_0 . . . 51
3.1076 so_22021524_22038330_0 . . . 51
3.1077 so_23060007_23060007_0 . . . 52
3.1078 so_23540764_23541097_1 . . . 52
3.1079 so_3632761_3633187_1 . . . 52
3.1080 so_29574616_29575182_4 . . . 52
3.1081 so_18099166_18100355_3 . . . 52
3.1082 so_21609006_22997514_0 . . . 52
3.1083 so_8514880_8514893_1 . . . 52
3.1084 so_18099166_18100355_4 . . . 52
3.1085 so_6120348_6120348_0 . . . 52
3.1086 so_23947055_23947055_0 . . . 52
3.1087 so_23926678_23926678_0 . . . 52
3.1088 so_27637768_27637768_0 . . . 52
3.1089 so_23976789_23976990_1 . . . 52
3.1090 so_6825801_6825801_1 . . . 52

3.1091 so_6825801_6825801_0 . . . 52
3.1092 so_23941480_23941480_0 . . . 52
3.1093 so_32235765_32604022_1 . . . 53
3.1094 so_25976267_25976453_2 . . . 53
3.1095 so_17826307_17826307_1 . . . 53
3.1096 so_6211198_6211198_0 . . . 53
3.1097 so_34800750_34801237_1 . . . 53
3.1098 so_7444790_7444919_2 . . . 53
3.1099 so_26821777_26821911_1 . . . 53
3.1100 so_24533091_24533452_4 . . . 53
3.1101 so_27481905_27483231_1 . . . 53
3.1102 so_24116817_24116818_0 . . . 53
3.1103 so_8111677_8111750_0 . . . 53
3.1104 so_42649844_42650066_0 . . . 53
3.1105 so_35373384_35375162_12 . . . 53
3.1106 so_9728693_9728764_0 . . . 53
3.1107 so_25925551_26005482_0 . . . 53
3.1108 so_21384604_21385079_4 . . . 53
3.1109 so_9819782_9819949_1 . . . 53
3.1110 so_9819782_9819949_2 . . . 53
3.1111 so_25133383_25133725_0 . . . 53
3.1112 so_1139063_1139083_0 . . . 53
3.1113 so_211376_211419_0 . . . 53
3.1114 so_46054228_46054902_5 . . . 53
3.1115 so_46054228_46054902_4 . . . 53
3.1116 so_47021238_47021276_1 . . . 53
3.1117 so_27178483_27179338_3 . . . 53
3.1118 so_27178483_27179338_4 . . . 53
3.1119 so_47020276_47020785_5 . . . 53
3.1120 so_32235765_32604022_8 . . . 53
3.1121 so_32235765_32604022_10 . . . 53
3.1122 so_25976267_25976453_3 . . . 53
3.1123 so_25976267_25976453_4 . . . 53
3.1124 so_37232481_37232747_1 . . . 53
3.1125 so_39959072_39959791_2 . . . 53
3.1126 so_39959072_39959791_4 . . . 53
3.1127 so_39959072_39959791_3 . . . 53
3.1128 so_42244250_42246348_3 . . . 53
3.1129 so_42244250_42246348_5 . . . 53
3.1130 so_42244250_42246348_6 . . . 53
3.1131 so_42244250_42246348_2 . . . 53
3.1132 so_27933307_27935812_1 . . . 53
3.1133 so_27764647_27765150_0 . . . 53
3.1134 so_27764647_27765150_1 . . . 53
3.1135 so_16752451_16773803_0 . . . 53
3.1136 so_11619122_11619310_0 . . . 53
3.1137 so_11619122_11619122_1 . . . 53
3.1138 so_40704924_40705005_1 . . . 53
3.1139 so_17996729_17996786_0 . . . 53
3.1140 so_27067433_27067700_4 . . . 53
3.1141 so_27067433_27067700_3 . . . 53

3.1142 so_42076314_42076537_0 . . . 53
3.1143 so_28187170_28188355_4 . . . 53
3.1144 so_20924599_20925002_0 . . . 53
3.1145 so_34075718_34075797_1 . . . 53
3.1146 so_34075718_34075797_0 . . . 53
3.1147 so_27481905_27483231_3 . . . 53
3.1148 so_42649844_42650066_2 . . . 53
3.1149 so_42649844_42650066_1 . . . 53
3.1150 so_29001446_29001446_1 . . . 53
3.1151 so_13192947_13193511_0 . . . 53
3.1152 so_18168207_18168207_0 . . . 53
3.1153 so_33959802_33961058_1 . . . 53
3.1154 so_33959802_33961058_0 . . . 53
3.1155 so_35373384_35375162_15 . . . 53
3.1156 so_35373384_35375162_13 . . . 53
3.1157 so_9728693_9728764_1 . . . 53
3.1158 so_25925551_26005482_1 . . . 53
3.1159 so_12920808_12921672_3 . . . 53
3.1160 so_12920808_12921672_1 . . . 53
3.1161 so_13325311_13325311_0 . . . 53
3.1162 so_11282437_11282663_1 . . . 53
3.1163 so_6855409_6855409_0 . . . 53
3.1164 so_6855409_6855409_2 . . . 53
3.1165 so_6855409_6855409_1 . . . 53
3.1166 so_6855409_6855409_3 . . . 53
3.1167 so_6855409_6855409_4 . . . 53
3.1168 so_34449254_34449471_1 . . . 53
3.1169 so_34449254_34449471_0 . . . 53
3.1170 so_46677029_46677339_1 . . . 53
3.1171 so_46677029_46677339_0 . . . 53
3.1172 so_46677029_46677339_2 . . . 53
3.1173 so_4227659_4227659_0 . . . 53
3.1174 so_6803556_6803635_2 . . . 53
3.1175 so_6803556_6803635_0 . . . 53
3.1176 so_19187766_19191910_0 . . . 53
3.1177 so_37027109_37028867_0 . . . 53
3.1178 so_40400836_40401633_0 . . . 53
3.1179 so_25874409_25903351_0 . . . 53
3.1180 so_27418384_27418628_1 . . . 53
3.1181 so_9819782_9819949_3 . . . 53
3.1182 so_9819782_9819949_4 . . . 53
3.1183 so_23603576_23603877_0 . . . 53
3.1184 so_3631672_3631724_3 . . . 53
3.1185 so_28242073_28242402_1 . . . 53
3.1186 so_33972793_33972870_0 . . . 53
3.1187 so_43530295_43552366_0 . . . 53
3.1188 so_19209453_19209453_1 . . . 53
3.1189 so_41689767_41690334_3 . . . 53
3.1190 so_41689767_41690334_1 . . . 53
3.1191 so_1139063_1139083_1 . . . 53
3.1192 so_3675829_3675851_0 . . . 53

3.1193 so_3675829_3675851_2 . . . 53
3.1194 so_27477963_27478480_0 . . . 53
3.1195 so_46992006_46992006_0 . . . 54
3.1196 so_33721762_33730436_0 . . . 54
3.1197 so_4396356_4397350_2 . . . 54
3.1198 so_4396356_4397350_3 . . . 54
3.1199 so_6922862_6922862_1 . . . 54
3.1200 so_6922862_6922862_2 . . . 54
3.1201 so_12170562_12205777_0 . . . 54
3.1202 so_35894955_35895314_1 . . . 54
3.1203 so_28035616_28059505_2 . . . 54
3.1204 so_28035616_28059505_0 . . . 54
3.1205 so_26821777_26821911_3 . . . 54
3.1206 so_26821777_26821911_2 . . . 54
3.1207 so_42392621_42392717_1 . . . 54
3.1208 so_42392621_42392717_0 . . . 54
3.1209 so_19802173_19802377_3 . . . 54
3.1210 so_19802173_19802377_4 . . . 54
3.1211 so_19802173_19802377_0 . . . 54
3.1212 so_19802173_19802377_2 . . . 54
3.1213 so_16696989_16697511_2 . . . 54
3.1214 so_16696989_16697511_1 . . . 54
3.1215 so_6803556_6803635_1 . . . 54
3.1216 so_13869150_19198730_0 . . . 54
3.1217 so_19209453_19209807_0 . . . 54
3.1218 so_31772588_31773348_0 . . . 54
3.1219 so_3675829_3675851_3 . . . 54
3.1220 so_37328444_37328923_1 . . . 55
3.1221 so_27544893_27545083_3 . . . 55
3.1222 so_11619122_11619122_3 . . . 55
3.1223 so_11619122_11619122_2 . . . 55
3.1224 so_7785886_7785935_1 . . . 55
3.1225 so_36025007_36025061_0 . . . 55
3.1226 so_36025007_36025061_2 . . . 55
3.1227 so_36025007_36025061_1 . . . 55
3.1228 so_36025007_36025061_3 . . . 55
3.1229 so_40489677_40489788_0 . . . 55
3.1230 so_25925551_26005482_4 . . . 55
3.1231 so_25925551_26005482_3 . . . 55
3.1232 so_27917680_27922878_6 . . . 55
3.1233 so_27917680_27922878_5 . . . 55
3.1234 so_27917680_27922878_4 . . . 55
3.1235 so_27418384_27418628_2 . . . 55
3.1236 so_3631672_3631724_4 . . . 55
3.1237 so_18311496_18311496_1 . . . 55
3.1238 so_40188047_40188186_1 . . . 55
3.1239 so_25633520_25633776_1 . . . 55
3.1240 so_24016605_24018720_0 . . . 55
3.1241 so_26744528_26744867_1 . . . 55
3.1242 so_21729819_21730048_0 . . . 55
3.1243 so_17957838_17958848_0 . . . 55

3.1244 so_5187891_5187935_1 . . . 55
3.1245 so_21290875_21291173_0 . . . 55
3.1246 so_6027955_6059889_0 . . . 55
3.1247 so_6027955_6059889_1 . . . 55
3.1248 so_28220331_28220408_0 . . . 55
3.1249 so_36779466_42440126_0 . . . 55
3.1250 so_24470404_30911330_0 . . . 55
3.1251 so_31464771_31464848_0 . . . 55
3.1252 so_47489881_47490236_1 . . . 55
3.1253 so_23994089_23994089_0 . . . 55
3.1254 so_29176179_29176486_1 . . . 55
3.1255 so_24095071_24095071_0 . . . 55
3.1256 so_30552141_30552236_0 . . . 55
3.1257 so_17437073_17437594_0 . . . 55
3.1258 so_21401944_21402811_1 . . . 55
3.1259 so_33966809_33967781_1 . . . 55
3.1260 so_42658852_42659101_0 . . . 55
3.1261 so_23976789_23976822_0 . . . 55
3.1262 so_41581225_41581251_0 . . . 55
3.1263 so_24104101_24104101_0 . . . 55
3.1264 so_24104101_24104274_0 . . . 55
3.1265 so_24104101_24104274_1 . . . 55
3.1266 so_24038003_24038178_1 . . . 55
3.1267 so_24038003_24038003_0 . . . 55
3.1268 so_24470404_24477408_0 . . . 55
3.1269 so_32362904_32362968_0 . . . 55
3.1270 so_44994461_44994784_2 . . . 55
3.1271 so_7701891_7701927_0 . . . 55
3.1272 so_17957838_17958848_1 . . . 55
3.1273 so_23991085_23991085_0 . . . 55
3.1274 so_75213_22085788_0 . . . 55
3.1275 so_5503901_5504296_0 . . . 55
3.1276 so_26744528_26744867_2 . . . 56
3.1277 so_27653959_27653999_0 . . . 56
3.1278 so_19060720_19061195_2 . . . 56
3.1279 so_30201414_30201574_0 . . . 56
3.1280 so_30201414_30201574_2 . . . 56
3.1281 so_24001273_24001273_1 . . . 56
3.1282 so_24001273_24001273_0 . . . 56
3.1283 so_47546616_47546665_0 . . . 56
3.1284 so_9668947_9668947_2 . . . 56
3.1285 so_24299963_24299963_0 . . . 56
3.1286 so_24293311_24313401_4 . . . 56
3.1287 so_46646773_46647030_1 . . . 56
3.1288 so_17957838_17958848_2 . . . 56
3.1289 so_10050863_10050964_1 . . . 56
3.1290 so_24095412_24095412_3 . . . 56
3.1291 so_24095412_24095412_0 . . . 56
3.1292 so_24095412_24095412_1 . . . 56
3.1293 so_31843161_31843161_2 . . . 56
3.1294 so_31843161_31843161_3 . . . 56

3.1295 so_31843161_31843161_4 . . . 56
3.1296 so_31843161_31843161_1 . . . 56
3.1297 so_31843161_31843161_0 . . . 56
3.1298 so_24373404_24375246_1 . . . 56
3.1299 so_24124493_24124493_1 . . . 56
3.1300 so_24124493_24124493_0 . . . 56
3.1301 so_19564009_19566059_2 . . . 56
3.1302 so_30552141_30552236_1 . . . 56
3.1303 so_11115484_11115950_3 . . . 56
3.1304 so_11115484_11115950_1 . . . 56
3.1305 so_11115484_11115950_4 . . . 56
3.1306 so_11115484_11115950_2 . . . 56
3.1307 so_11115484_11115950_5 . . . 56
3.1308 so_29855645_29855645_1 . . . 57
3.1309 so_29855645_29855645_0 . . . 57
3.1310 so_29855645_29855645_2 . . . 57
3.1311 so_12843893_12844039_1 . . . 57
3.1312 so_37652083_37652168_1 . . . 57
3.1313 so_37652083_37652168_2 . . . 57
3.1314 so_37652083_37652168_4 . . . 57
3.1315 so_7394492_7408582_0 . . . 57
3.1316 so_7394492_7408582_1 . . . 57
3.1317 so_34671852_34671852_8 . . . 57
3.1318 so_34671852_34671852_5 . . . 57
3.1319 so_34671852_34671852_7 . . . 57
3.1320 so_34671852_34671852_6 . . . 57
3.1321 so_34671852_34671852_9 . . . 57
3.1322 so_30554119_30554202_5 . . . 57
3.1323 so_30554119_30554202_3 . . . 57
3.1324 so_30554119_30554202_4 . . . 57
3.1325 so_30554119_30554202_6 . . . 57
3.1326 so_10050863_10050964_4 . . . 57
3.1327 so_10050863_10050964_3 . . . 57
3.1328 so_46412754_46412864_0 . . . 57
3.1329 so_46412754_46412864_1 . . . 57
3.1330 so_28220331_28220331_0 . . . 57
3.1331 so_29705794_29705836_2 . . . 57
3.1332 so_29705794_29705836_1 . . . 57
3.1333 so_27954940_28511153_2 . . . 57
3.1334 so_27954940_28511153_3 . . . 57
3.1335 so_27954940_28511153_1 . . . 57
3.1336 so_27954940_28511153_4 . . . 57
3.1337 so_44295575_44314006_0 . . . 57
3.1338 so_24373404_24373404_0 . . . 57
3.1339 so_2823485_2824067_0 . . . 57
3.1340 so_26558507_26682739_1 . . . 57
3.1341 so_26558507_26682739_6 . . . 57
3.1342 so_26558507_26682739_9 . . . 57
3.1343 so_26558507_26682739_2 . . . 57
3.1344 so_26558507_26682739_5 . . . 57
3.1345 so_26558507_26682739_4 . . . 57

3.1346 so_26558507_26682739_7 . . . 57
3.1347 so_26558507_26682739_10 . . . 57
3.1348 so_14492878_14493136_3 . . . 57
3.1349 so_14492878_14493136_1 . . . 57
3.1350 so_14492878_14493136_2 . . . 57
3.1351 so_31769853_31769853_4 . . . 57
3.1352 so_4420828_4420839_3 . . . 57
3.1353 so_32006122_32006973_3 . . . 57
3.1354 so_32006122_32006973_0 . . . 57
3.1355 so_32006122_32006973_1 . . . 57
3.1356 so_12883486_12883770_0 . . . 57
3.1357 so_25541010_25541010_0 . . . 57
3.1358 so_25541010_25541010_1 . . . 57
3.1359 so_25541010_25541010_3 . . . 57
3.1360 so_25541010_25541197_1 . . . 57
3.1361 so_31129781_31163087_2 . . . 57
3.1362 so_31129781_31163087_1 . . . 57
3.1363 so_25321859_25321859_0 . . . 57
3.1364 so_7006938_7006938_1 . . . 57
3.1365 so_7006938_7006938_0 . . . 57
3.1366 so_37301946_37303140_0 . . . 57
3.1367 so_7006968_7006968_0 . . . 57
3.1368 so_27903704_27903879_2 . . . 57
3.1369 so_28163389_28163389_0 . . . 57
3.1370 so_24043802_24043802_0 . . . 57
3.1371 so_24268240_24268240_0 . . . 57
3.1372 so_27252466_27252746_0 . . . 57
3.1373 so_33313853_33318306_1 . . . 57
3.1374 so_28453118_28453118_0 . . . 57
3.1375 so_26070916_26071134_3 . . . 57
3.1376 so_31628853_31629062_0 . . . 57
3.1377 so_31628853_31629071_1 . . . 57
3.1378 so_8349859_8350189_0 . . . 57
3.1379 so_14235780_14236302_0 . . . 57
3.1380 so_13078893_13079051_3 . . . 57
3.1381 so_37301946_37303140_1 . . . 57
3.1382 so_24268165_24268165_1 . . . 57
3.1383 so_24328888_24329404_0 . . . 57
3.1384 so_24930921_24931369_0 . . . 57
3.1385 so_24261826_24261826_0 . . . 57
3.1386 so_24268240_24268240_1 . . . 57
3.1387 so_24268240_24268240_2 . . . 57
3.1388 so_27252466_27252746_1 . . . 57
3.1389 so_37301946_37303140_2 . . . 57
3.1390 so_24268165_24268165_2 . . . 57
3.1391 so_24268165_24268165_0 . . . 57
3.1392 so_24328888_24328888_0 . . . 57
3.1393 so_24328888_24328888_2 . . . 57
3.1394 so_25176985_25177404_0 . . . 60
3.1395 so_25176985_25177404_1 . . . 60
3.1396 so_4094008_4094068_1 . . . 61

3.1397 so_26755400_26758377_0 . . . 61
3.1398 so_20814505_20814545_1 . . . 61
3.1399 so_4163799_4163806_2 . . . 61
3.1400 so_15317218_15317235_0 . . . 61
3.1401 so_26168445_26169292_1 . . . 61
3.1402 so_26168445_26169292_3 . . . 61
3.1403 so_27890699_27890824_0 . . . 61
3.1404 so_29827850_29828989_0 . . . 61
3.1405 so_40257011_40257111_3 . . . 61
3.1406 so_40257011_40257321_4 . . . 61
3.1407 so_40257011_40257111_1 . . . 61
3.1408 so_40257011_40257321_3 . . . 61
3.1409 so_31875710_31876176_8 . . . 61
3.1410 so_31875710_31876176_10 . . . 61
3.1411 so_29198354_29198354_9 . . . 61
3.1412 so_29198354_29198354_11 . . . 61
3.1413 so_29198354_29198354_7 . . . 61
3.1414 so_4420828_4420839_1 . . . 61
3.1415 so_15317218_15317235_1 . . . 61
3.1416 so_8070233_8070789_1 . . . 61
3.1417 so_37219984_39203657_4 . . . 61
3.1418 so_37219984_39203657_0 . . . 61
3.1419 so_37219984_39203657_2 . . . 61
3.1420 so_41471439_41472929_0 . . . 61
3.1421 so_41471439_41472929_2 . . . 61
3.1422 so_37744047_37744321_1 . . . 61
3.1423 so_37744047_37744321_0 . . . 61
3.1424 so_10426428_10426708_0 . . . 62
3.1425 so_15064535_15066266_0 . . . 62
3.1426 so_15064535_15066266_4 . . . 62
3.1427 so_30011740_30014205_0 . . . 62
3.1428 so_27930448_28021419_18 . . . 62
3.1429 so_40470650_40471868_1 . . . 62
3.1430 so_31690932_31752995_1 . . . 62
3.1431 so_31690932_31756207_2 . . . 62
3.1432 so_10671406_10676746_1 . . . 62
3.1433 so_10671406_10806458_1 . . . 62
3.1434 so_35997875_35999952_1 . . . 62
3.1435 so_24358606_24358606_0 . . . 62
3.1436 so_24358606_24358853_1 . . . 62
3.1437 so_32665178_32665689_0 . . . 62
3.1438 so_7228957_7228957_0 . . . 62
3.1439 so_6845779_6846266_0 . . . 62
3.1440 so_24593942_24594393_1 . . . 62
3.1441 so_25376209_25376509_1 . . . 62
3.1442 so_31769853_31769853_2 . . . 62
3.1443 so_14156640_14157511_0 . . . 62
3.1444 so_18206923_18207975_0 . . . 62
3.1445 so_18206923_18206923_1 . . . 62
3.1446 so_18218475_18218475_0 . . . 62
3.1447 so_26243781_26243781_0 . . . 62

3.1448 so_26243781_26243781_2 . . . 62
3.1449 so_11115484_11116164_0 . . . 62
3.1450 so_8100492_8453351_0 . . . 62
3.1451 so_29583735_29583735_0 . . . 63
3.1452 so_18218475_18218475_1 . . . 63
3.1453 so_37310866_37311097_0 . . . 63
3.1454 so_24107629_24107629_0 . . . 63
3.1455 so_40658511_40658720_0 . . . 63
3.1456 so_18242603_18242603_0 . . . 63
3.1457 so_44475056_44475329_0 . . . 63
3.1458 so_26058966_26058966_0 . . . 63
3.1459 Program and linkage . . . 63
3.1460 so_16506196_25634172_2 . . . 63
3.1461 so_30015149_30015191_0 . . . 63
3.1462 so_31191719_31191780_0 . . . 63
3.1463 so_8452952_24383995_1 . . . 63
3.1464 so_8452952_24383995_3 . . . 63
3.1465 so_8452952_24383995_2 . . . 63
3.1466 so_8452952_24383995_4 . . . 63
3.1467 so_18498957_18499719_0 . . . 63
3.1468 so_3341055_3341707_1 . . . 63
3.1469 so_3341055_3341707_0 . . . 63
3.1470 so_27541028_27541143_2 . . . 63
3.1471 so_27541028_27541143_1 . . . 63
3.1472 so_27541028_27541143_0 . . . 63
3.1473 so_23645495_23645495_0 . . . 63
3.1474 so_23645495_23645786_0 . . . 63
3.1475 so_23645495_23645495_2 . . . 63
3.1476 so_37442606_37442607_0 . . . 63
3.1477 so_22867654_23087392_0 . . . 63
3.1478 so_22867654_23087392_2 . . . 63
3.1479 so_22867654_23087392_3 . . . 63
3.1480 so_22867654_23087392_1 . . . 63
3.1481 so_13898053_13898517_2 . . . 63
3.1482 so_29224679_29224988_0 . . . 63
3.1483 so_17090354_17099819_0 . . . 63
3.1484 so_17090354_17099819_1 . . . 63
3.1485 so_35288838_35290352_1 . . . 63
3.1486 so_35288838_35290352_0 . . . 63
3.1487 so_36841401_36841449_0 . . . 63
3.1488 so_45160517_45160517_2 . . . 63
3.1489 so_45160517_45160517_1 . . . 63
3.1490 so_6465325_26052100_0 . . . 63
3.1491 so_9969156_9969210_1 . . . 63
3.1492 so_8069717_8069831_1 . . . 63
3.1493 so_185624_189162_3 . . . 63
3.1494 so_4185687_4185709_0 . . . 63
3.1495 so_4185687_4185709_1 . . . 63
3.1496 so_8452952_24383995_0 . . . 63
3.1497 so_22765649_22765765_0 . . . 63
3.1498 so_22765649_22765649_0 . . . 63

3.1499 so_22765649_22765649_2 . . . 63
3.1500 so_13615046_13615131_0 . . . 63
3.1501 so_23479015_23479190_1 . . . 63
3.1502 so_15119932_15119968_0 . . . 63
3.1503 so_3538807_3538807_0 . . . 63
3.1504 so_30517546_30517574_6 . . . 63
3.1505 so_30517546_30517574_5 . . . 63
3.1506 so_30517546_30517574_4 . . . 63
3.1507 so_21065112_21066022_0 . . . 63
3.1508 so_21065112_21066022_2 . . . 63
3.1509 so_21065112_21066022_1 . . . 63
3.1510 so_21065112_21066022_4 . . . 63
3.1511 so_21065112_21066022_5 . . . 63
3.1512 so_20716091_20716155_4 . . . 63
3.1513 so_20716091_20716155_2 . . . 63
3.1514 so_14616183_14616201_0 . . . 63
3.1515 so_23645495_23645495_1 . . . 63
3.1516 so_6318779_6318819_0 . . . 63
3.1517 so_28106468_28106535_1 . . . 63
3.1518 so_28106468_28106535_0 . . . 63
3.1519 so_994353_994428_2 . . . 63
3.1520 so_994353_994428_4 . . . 63
3.1521 so_20231616_20232726_0 . . . 63
3.1522 so_21076225_21076378_0 . . . 63
3.1523 so_28766075_28766204_1 . . . 63
3.1524 so_13673864_13673954_0 . . . 63
3.1525 so_47421818_47421861_0 . . . 63
3.1526 so_47421818_47421861_1 . . . 63
3.1527 so_22867654_23087392_4 . . . 63
3.1528 so_22867654_23087392_6 . . . 63
3.1529 so_22867654_23087392_5 . . . 63
3.1530 so_35376814_35376814_1 . . . 63
3.1531 so_46107312_46107312_0 . . . 63
3.1532 so_30183183_30183556_1 . . . 63
3.1533 so_30183183_30183556_0 . . . 63
3.1534 so_26290383_26290498_0 . . . 63
3.1535 so_20840556_20840945_1 . . . 63
3.1536 so_33376124_33376124_0 . . . 63
3.1537 so_33376124_33376124_2 . . . 63
3.1538 so_33376124_33376124_1 . . . 63
3.1539 so_37592419_37593204_0 . . . 63
3.1540 so_4181059_4181132_0 . . . 63
3.1541 so_4181059_4181132_2 . . . 63
3.1542 so_4181059_4181132_1 . . . 63
3.1543 so_30604686_30604988_4 . . . 63
3.1544 so_30604686_30604988_2 . . . 63
3.1545 so_12701130_12701279_1 . . . 63
3.1546 so_12701130_12701279_0 . . . 63
3.1547 so_28761549_28761757_0 . . . 63
3.1548 so_28761549_28761757_1 . . . 63
3.1549 so_5787466_5787490_0 . . . 63

3.1550 so_45160517_45160517_3 . . . 63
3.1551 so_45160517_45160517_4 . . . 63
3.1552 so_6465325_26052100_1 . . . 63
3.1553 so_24558413_24558597_0 . . . 63
3.1554 so_6173872_6173889_0 . . . 63
3.1555 so_39933440_39933440_3 . . . 63
3.1556 so_9969156_9969210_0 . . . 63
3.1557 so_14403796_14404206_0 . . . 63
3.1558 so_8001047_8001177_0 . . . 63
3.1559 so_8001047_8001177_1 . . . 63
3.1560 so_5261299_5261430_1 . . . 63
3.1561 so_185624_189162_1 . . . 63
3.1562 so_185624_189162_0 . . . 63
3.1563 so_45358729_45359202_1 . . . 63
3.1564 so_7159235_7160293_0 . . . 63
3.1565 so_7159235_7160293_1 . . . 63
3.1566 so_31129919_31130300_0 . . . 63
3.1567 so_33402177_33402258_0 . . . 63
3.1568 so_33402177_33402258_1 . . . 63
3.1569 so_23479015_23479190_0 . . . 64
3.1570 so_25788127_31652888_2 . . . 64
3.1571 so_26085345_26085345_0 . . . 64
3.1572 so_154469_156834_2 . . . 64
3.1573 so_26529842_26530462_2 . . . 64
3.1574 so_37442606_37442607_2 . . . 64
3.1575 so_8339562_8339562_0 . . . 64
3.1576 so_26026273_26026273_0 . . . 64
3.1577 so_33377899_33377899_0 . . . 64
3.1578 so_13673864_13673954_1 . . . 64
3.1579 so_14909997_14909999_15 . . . 64
3.1580 so_14909997_14909999_11 . . . 64
3.1581 so_14909997_14909999_16 . . . 64
3.1582 so_14909997_14909999_14 . . . 64
3.1583 so_35376814_35376814_0 . . . 64
3.1584 so_35376814_35376814_2 . . . 64
3.1585 so_41167119_41200573_2 . . . 64
3.1586 so_41167119_41200573_0 . . . 64
3.1587 so_4181059_19501842_5 . . . 64
3.1588 so_4181059_19501842_3 . . . 64
3.1589 so_4181059_19501842_7 . . . 64
3.1590 so_4181059_19501842_11 . . . 64
3.1591 so_4181059_19501842_0 . . . 64
3.1592 so_4181059_19501842_4 . . . 64
3.1593 so_4181059_19501842_10 . . . 64
3.1594 so_4181059_19501842_12 . . . 64
3.1595 so_35288838_35289169_1 . . . 64
3.1596 so_35288838_35288838_0 . . . 64
3.1597 so_908830_910686_3 . . . 64
3.1598 so_28761549_28761757_2 . . . 64
3.1599 so_28904035_28904035_0 . . . 64
3.1600 so_6465325_26052100_3 . . . 64

3.1601 so_6465325_26052100_5 . . . 64
3.1602 so_6465325_26052100_7 . . . 64
3.1603 so_39933440_39933440_4 . . . 64
3.1604 so_16922373_16925113_0 . . . 64
3.1605 so_10022355_10022400_0 . . . 64
3.1606 so_42555541_42555541_0 . . . 64
3.1607 so_4866290_4866383_1 . . . 64
3.1608 so_1255460_1255659_2 . . . 64
3.1609 so_908830_910686_2 . . . 64
3.1610 so_42376853_42376853_0 . . . 64
3.1611 so_26085345_26085345_3 . . . 64
3.1612 so_23367912_23368809_0 . . . 64
3.1613 so_24771192_24771913_0 . . . 64
3.1614 so_24239846_24250341_2 . . . 64
3.1615 so_24239846_24250341_3 . . . 64
3.1616 so_26026273_26026610_0 . . . 64
3.1617 so_38588167_38589645_0 . . . 64
3.1618 so_46779747_46779748_0 . . . 64
3.1619 so_26616163_26616558_0 . . . 64
3.1620 so_16039723_16039723_0 . . . 64
3.1621 so_17090354_17099819_2 . . . 64
3.1622 so_31493734_31493944_0 . . . 64
3.1623 so_23988716_23988881_1 . . . 64
3.1624 so_41978949_41978949_0 . . . 64
3.1625 so_8069717_8069831_2 . . . 64
3.1626 so_29904114_29904114_0 . . . 64
3.1627 so_29904114_29904114_1 . . . 64
3.1628 so_24239846_24250341_5 . . . 64
3.1629 so_24239846_24250341_6 . . . 64
3.1630 so_14144953_14145071_0 . . . 64
3.1631 so_29928957_29929143_0 . . . 64
3.1632 so_29928957_29929153_0 . . . 64
3.1633 so_26616163_26616558_1 . . . 64
3.1634 so_36048089_36048365_0 . . . 64
3.1635 so_26067473_26069040_1 . . . 64
3.1636 so_14597463_14597463_0 . . . 65
3.1637 so_35370521_35370521_0 . . . 65
3.1638 so_35370521_35370521_1 . . . 65
3.1639 so_18498957_18500500_0 . . . 65
3.1640 so_18498957_18500500_1 . . . 65
3.1641 so_5751977_5752000_0 . . . 65
3.1642 so_5751977_5751990_1 . . . 65
3.1643 so_12355161_12355267_0 . . . 65
3.1644 so_24018932_24019645_0 . . . 65
3.1645 so_24018932_24019645_1 . . . 65
3.1646 so_5720359_5720418_0 . . . 65
3.1647 so_2598178_2598272_0 . . . 65
3.1648 so_34699569_34699606_0 . . . 65
3.1649 so_8814146_8814164_0 . . . 65
3.1650 so_8814146_8814167_0 . . . 65
3.1651 so_3184939_3185008_0 . . . 65

3.1652 so_991062_991162_0 . . . 65
3.1653 so_18994041_18997863_1 . . . 65
3.1654 so_2036892_2036902_0 . . . 65
3.1655 so_4569928_4569975_0 . . . 65
3.1656 so_5131691_5131691_0 . . . 65
3.1657 so_3485472_24560240_0 . . . 65
3.1658 so_33420347_33421449_7 . . . 65
3.1659 so_33420347_33421449_3 . . . 65
3.1660 so_33420347_33420347_0 . . . 65
3.1661 so_33420347_33421449_5 . . . 65
3.1662 so_33420347_33421449_8 . . . 65
3.1663 so_33420347_33421449_10 . . . 65
3.1664 so_3413044_3413155_0 . . . 65
3.1665 so_742607_742617_1 . . . 65
3.1666 so_21201685_21201894_0 . . . 65
3.1667 so_35344795_35344795_0 . . . 65
3.1668 so_17013534_17014155_0 . . . 65
3.1669 so_13667057_13703139_0 . . . 65
3.1670 so_6465325_26052100_4 . . . 65
3.1671 so_6465325_26052100_9 . . . 65
3.1672 so_6465325_26052100_10 . . . 65
3.1673 so_6465325_26052100_8 . . . 65
3.1674 so_39933440_39933440_2 . . . 65
3.1675 so_39933440_39933440_1 . . . 65
3.1676 so_26749011_26749031_1 . . . 65
3.1677 so_32326444_32326577_0 . . . 65
3.1678 so_12045029_12045179_0 . . . 65
3.1679 so_2152845_2152910_0 . . . 65
3.1680 so_6555826_6555871_0 . . . 65
3.1681 so_876048_876069_0 . . . 65
3.1682 so_2662843_2662848_0 . . . 65
3.1683 so_8203750_8203922_0 . . . 65
3.1684 so_8203750_8203860_0 . . . 65
3.1685 so_3172909_3172912_0 . . . 65
3.1686 so_24877338_24877581_3 . . . 65
3.1687 so_12978745_12979016_0 . . . 66
3.1688 so_12978745_12979016_2 . . . 66
3.1689 so_12978745_12979016_1 . . . 66
3.1690 so_12978745_12979016_3 . . . 66
3.1691 so_12978745_12979016_4 . . . 66
3.1692 so_41539456_41539584_3 . . . 66
3.1693 so_35376814_35377004_0 . . . 66
3.1694 so_35376814_35376814_3 . . . 66
3.1695 so_35376814_35377004_3 . . . 66
3.1696 so_35376814_35377004_4 . . . 66
3.1697 so_35376814_35377004_1 . . . 66
3.1698 so_35376814_35377004_2 . . . 66
3.1699 so_14420908_14420908_3 . . . 66
3.1700 so_14420908_14420908_4 . . . 66
3.1701 so_14420908_14420908_2 . . . 66
3.1702 so_14420908_14420908_0 . . . 66

3.1703 so_14420908_14420908_1 . . . 66
3.1704 so_26067473_26069198_1 . . . 66
3.1705 so_27221737_27222043_0 . . . 66
3.1706 so_18547537_18547665_0 . . . 66
3.1707 so_42555541_42555541_1 . . . 66
3.1708 so_26009080_26010140_0 . . . 66
3.1709 so_23974070_23980018_0 . . . 66
3.1710 so_43441714_43441944_0 . . . 66
3.1711 so_6011517_6011517_0 . . . 66
3.1712 so_12485025_12485165_0 . . . 66
3.1713 so_2068022_2068048_1 . . . 66
3.1714 so_24452139_24452139_0 . . . 66
3.1715 so_36048089_36048365_1 . . . 66
3.1716 so_4810664_7439261_1 . . . 66
3.1717 so_4810664_7439261_0 . . . 66
3.1718 so_28835198_28836383_0 . . . 66
3.1719 so_26386010_26386337_2 . . . 66
3.1720 so_2068022_2068048_2 . . . 66
3.1721 Start and termination . . . 66
3.1722 so_32807964_32808119_0 . . . 66
3.1723 so_26023993_26024133_0 . . . 66
3.1724 so_32851184_32851722_1 . . . 66
3.1725 so_32851184_32851348_0 . . . 66
3.1726 so_32851184_32851400_1 . . . 66
3.1727 so_20218690_20218759_2 . . . 66
3.1728 so_20218690_20218759_0 . . . 66
3.1729 so_12004258_12004947_0 . . . 66
3.1730 so_10155045_10155139_0 . . . 66
3.1731 so_15081553_15081563_0 . . . 66
3.1732 so_22710903_22711108_0 . . . 66
3.1733 so_1621574_1621664_1 . . . 66
3.1734 so_14794547_14794559_0 . . . 66
3.1735 so_17515790_17515861_0 . . . 66
3.1736 so_5296163_14831226_0 . . . 66
3.1737 so_204476_18721336_8 . . . 66
3.1738 so_204476_31263079_15 . . . 66
3.1739 so_39789054_39789172_1 . . . 66
3.1740 so_6546669_6546698_0 . . . 66
3.1741 so_7851427_7851503_0 . . . 66
3.1742 so_7851427_7851565_0 . . . 66
3.1743 so_42441683_42442198_3 . . . 66
3.1744 so_33212374_33212391_0 . . . 66
3.1745 so_22614449_22614761_1 . . . 66
3.1746 so_13094090_13094171_4 . . . 66
3.1747 so_3974796_3974896_0 . . . 66
3.1748 so_25471434_25472906_0 . . . 66
3.1749 so_2626533_2626616_0 . . . 66
3.1750 so_30271822_30275587_0 . . . 66
3.1751 so_16781458_16781482_0 . . . 66
3.1752 so_16781458_16781475_0 . . . 66
3.1753 so_28484557_28484628_0 . . . 66

3.1754 so_25630829_25631038_1 . . . 66
3.1755 so_19419569_19419617_9 . . . 66
3.1756 so_23553723_23553744_1 . . . 66
3.1757 so_23553723_23553808_0 . . . 66
3.1758 so_3956678_3956683_0 . . . 66
3.1759 so_24464164_24464164_0 . . . 66
3.1760 so_4783404_4783404_0 . . . 66
3.1761 so_4783404_4784365_0 . . . 66
3.1762 so_7050925_7050940_0 . . . 66
3.1763 so_3831200_3831216_0 . . . 66
3.1764 so_39106798_39106982_0 . . . 66
3.1765 so_39106798_39106983_0 . . . 66
3.1766 so_39106798_39106983_1 . . . 66
3.1767 so_39106798_39107034_0 . . . 66
3.1768 so_38768582_38768612_0 . . . 66
3.1769 so_9857048_9857066_0 . . . 66
3.1770 so_24232937_24233435_1 . . . 66
3.1771 so_18547114_18547129_2 . . . 66
3.1772 so_18547114_18547129_1 . . . 66
3.1773 so_17124115_17124189_0 . . . 66
3.1774 so_1915659_1915659_0 . . . 66
3.1775 so_1915659_1915695_0 . . . 66
3.1776 so_17393842_17393943_0 . . . 66
3.1777 so_17393842_17393943_2 . . . 66
3.1778 so_40966790_40966845_0 . . . 66
3.1779 so_30850094_30850121_0 . . . 66
3.1780 so_18912852_18912924_1 . . . 66
3.1781 so_28125290_28125378_1 . . . 66
3.1782 so_22710903_22711108_1 . . . 66
3.1783 so_22710903_22710975_0 . . . 66
3.1784 so_27367051_27367051_3 . . . 66
3.1785 so_27367051_27367051_2 . . . 66
3.1786 so_27367051_27367051_0 . . . 66
3.1787 so_42875426_42875557_0 . . . 66
3.1788 so_47583864_47584093_2 . . . 66
3.1789 so_17863494_17863556_0 . . . 66
3.1790 so_8146647_8146736_0 . . . 66
3.1791 so_1621574_1621664_0 . . . 66
3.1792 so_37873874_37874013_0 . . . 66
3.1793 so_37873874_37873874_1 . . . 66
3.1794 so_37873874_37873874_2 . . . 66
3.1795 so_37873874_37874305_2 . . . 66
3.1796 so_29498165_29498206_0 . . . 66
3.1797 so_5296163_14831226_1 . . . 66
3.1798 so_14513534_14513705_0 . . . 66
3.1799 so_31112154_31112370_0 . . . 66
3.1800 so_204476_18721336_9 . . . 66
3.1801 so_204476_31263079_16 . . . 66
3.1802 so_204476_31263079_18 . . . 66
3.1803 so_204476_18721336_11 . . . 66
3.1804 so_204476_31263079_13 . . . 66

3.1805 so_39789054_39789172_2 . . . 66
3.1806 so_11355513_11355557_0 . . . 66
3.1807 so_19219363_19219421_1 . . . 66
3.1808 so_6408183_6408230_0 . . . 66
3.1809 so_6408183_6408272_0 . . . 66
3.1810 so_6408183_6408272_1 . . . 66
3.1811 so_10366022_10366093_0 . . . 66
3.1812 so_10366022_10366093_1 . . . 66
3.1813 so_2525183_2525185_0 . . . 66
3.1814 so_21968468_21968530_2 . . . 66
3.1815 so_21968468_21968530_0 . . . 66
3.1816 so_5282151_5282153_0 . . . 66
3.1817 so_44859450_44859591_4 . . . 66
3.1818 so_44859450_44859591_2 . . . 66
3.1819 so_36945552_36959404_3 . . . 66
3.1820 so_18988670_18988695_0 . . . 66
3.1821 so_22239_22262_1 . . . 66
3.1822 so_38195245_38195440_1 . . . 66
3.1823 so_27367400_27367400_0 . . . 66
3.1824 so_27367400_27367400_2 . . . 66
3.1825 so_35011017_35011317_0 . . . 66
3.1826 so_35011017_35011317_2 . . . 66
3.1827 so_26470912_26470912_0 . . . 66
3.1828 so_26470912_26470912_2 . . . 66
3.1829 so_26470912_26471420_0 . . . 66
3.1830 so_43885942_43885943_0 . . . 66
3.1831 so_16942250_16942441_2 . . . 66
3.1832 so_16942250_16942441_6 . . . 66
3.1833 so_10821417_10821475_1 . . . 66
3.1834 so_4207134_4211927_1 . . . 66
3.1835 so_4207134_4211927_0 . . . 66
3.1836 so_4207134_44859345_2 . . . 66
3.1837 so_45734074_45734561_0 . . . 66
3.1838 so_40099023_40099078_1 . . . 66
3.1839 so_22614449_22614761_2 . . . 66
3.1840 so_22614449_22614761_4 . . . 66
3.1841 so_22614449_22614761_3 . . . 66
3.1842 so_13094090_13094127_0 . . . 66
3.1843 so_11216155_11216266_0 . . . 66
3.1844 so_4031672_4031745_2 . . . 66
3.1845 so_15094828_15094867_2 . . . 66
3.1846 so_15094828_15094867_4 . . . 66
3.1847 so_15094828_15094867_0 . . . 66
3.1848 so_2336027_2336110_0 . . . 66
3.1849 so_17134975_17135093_4 . . . 66
3.1850 so_32659640_32660154_4 . . . 66
3.1851 so_39203571_39203738_0 . . . 66
3.1852 so_9984188_9985582_0 . . . 66
3.1853 so_8844915_8845367_0 . . . 66
3.1854 so_19419569_19419617_10 . . . 66
3.1855 so_7226044_7226081_0 . . . 66

3.1856 so_383973_383976_0 . . . 66
3.1857 so_383973_383976_2 . . . 66
3.1858 so_19803522_19803545_1 . . . 66
3.1859 so_4134992_4135041_0 . . . 66
3.1860 so_14450653_14450687_2 . . . 66
3.1861 so_14450653_14450687_0 . . . 66
3.1862 so_3652182_3652214_1 . . . 66
3.1863 so_3652182_3652214_0 . . . 66
3.1864 so_27139519_27139549_0 . . . 66
3.1865 so_2050961_2051026_0 . . . 66
3.1866 so_32194151_32194324_1 . . . 66
3.1867 so_32194151_32194256_1 . . . 66
3.1868 so_32194151_32194324_0 . . . 66
3.1869 so_27419071_27419648_4 . . . 66
3.1870 so_3934769_3938789_0 . . . 66
3.1871 so_27777071_27777306_0 . . . 67
3.1872 so_32851184_32851348_1 . . . 67
3.1873 so_31222601_31222782_0 . . . 67
3.1874 so_17620241_17620328_0 . . . 67
3.1875 so_15081553_15081619_0 . . . 67
3.1876 so_47583864_47584093_3 . . . 67
3.1877 so_21539744_21539788_0 . . . 67
3.1878 so_33015748_33015952_0 . . . 67
3.1879 so_6221953_6221987_0 . . . 67
3.1880 so_204476_18721336_12 . . . 67
3.1881 so_36363704_36363740_0 . . . 67
3.1882 so_5326491_5326547_0 . . . 67
3.1883 so_5759039_5759114_0 . . . 67
3.1884 so_18460912_18461000_1 . . . 67
3.1885 so_6981104_6982101_0 . . . 67
3.1886 so_16719234_16719234_0 . . . 67
3.1887 so_22614449_22614761_5 . . . 67
3.1888 so_13094090_13094127_1 . . . 67
3.1889 so_4518598_4518651_0 . . . 67
3.1890 so_27419071_27419648_8 . . . 67
3.1891 so_30488692_30488692_0 . . . 67
3.1892 so_30250934_30251056_8 . . . 67
3.1893 so_2668075_2668101_0 . . . 67
3.1894 so_2668075_2668098_1 . . . 67
3.1895 so_7372937_7372968_0 . . . 67
3.1896 so_22614449_22614761_6 . . . 67
3.1897 so_17149613_17149837_0 . . . 67
3.1898 so_19293642_19293663_0 . . . 67
3.1899 so_38768582_38768612_3 . . . 67
3.1900 so_1610030_1610454_1 . . . 67
3.1901 so_29974523_29974618_0 . . . 67
3.1902 so_29974523_29974540_0 . . . 67
3.1903 so_22586250_22586378_0 . . . 67
3.1904 so_7255644_7255794_1 . . . 67
3.1905 so_8696698_8696722_0 . . . 67
3.1906 so_17551087_17551625_0 . . . 67

3.1907 so_3459810_3459887_1 . . . 67
3.1908 so_15911517_15911747_1 . . . 67
3.1909 so_37873874_37874013_2 . . . 67
3.1910 so_204476_18721336_13 . . . 67
3.1911 so_204476_43558724_1 . . . 67
3.1912 so_18402853_18402953_0 . . . 67
3.1913 so_18402853_18402912_1 . . . 67
3.1914 so_18402853_18402983_0 . . . 67
3.1915 so_19219363_19219421_3 . . . 67
3.1916 so_14853114_14853204_0 . . . 67
3.1917 so_7920531_7920563_1 . . . 67
3.1918 so_15474512_15474559_2 . . . 67
3.1919 so_276807_276814_0 . . . 67
3.1920 so_22239_22262_4 . . . 67
3.1921 so_24074392_24074411_1 . . . 67
3.1922 so_27367400_27367400_3 . . . 67
3.1923 so_39026390_39026917_0 . . . 67
3.1924 so_39026390_39027129_0 . . . 67
3.1925 so_26470912_26470912_3 . . . 67
3.1926 so_13545291_13545291_0 . . . 67
3.1927 so_24091108_24091117_0 . . . 67
3.1928 so_16719234_16719290_0 . . . 67
3.1929 so_4207134_8697019_0 . . . 67
3.1930 so_22614449_22614761_7 . . . 67
3.1931 so_17134975_17135093_1 . . . 67
3.1932 so_39203571_39203738_3 . . . 67
3.1933 so_16568920_16568969_0 . . . 67
3.1934 so_34912588_34913061_0 . . . 67
3.1935 so_17516560_17516581_0 . . . 67
3.1936 so_12512177_12512811_0 . . . 67
3.1937 so_3379843_3379857_0 . . . 67
3.1938 so_24701438_24701438_0 . . . 67
3.1939 so_7923624_7925027_0 . . . 67
3.1940 so_20788960_20789142_1 . . . 67
3.1941 so_35886358_35886494_2 . . . 67
3.1942 so_7074647_7074714_1 . . . 67
3.1943 so_22783166_22783340_1 . . . 67
3.1944 so_41596068_41596068_1 . . . 67
3.1945 so_23128045_23128045_3 . . . 67
3.1946 so_23555612_23558302_1 . . . 67
3.1947 so_38386993_38387362_5 . . . 67
3.1948 so_38386993_38387362_2 . . . 67
3.1949 so_28563974_28564493_0 . . . 67
3.1950 so_3514991_3515026_0 . . . 67
3.1951 so_22414406_22414665_1 . . . 67
3.1952 so_22414406_22414665_0 . . . 67
3.1953 so_27167958_27168696_7 . . . 67
3.1954 so_24572870_24572870_0 . . . 67
3.1955 so_6915_6955_0 . . . 67
3.1956 so_22021295_22021934_0 . . . 67
3.1957 so_21328655_21328691_0 . . . 67

3.1958 so_24253799_24253832_4 . . . 67
3.1959 so_24253799_24253832_3 . . . 67
3.1960 so_25957705_25957705_0 . . . 67
3.1961 so_40138325_40138652_1 . . . 67
3.1962 so_25470791_25470791_5 . . . 67
3.1963 so_25470791_25470791_3 . . . 67
3.1964 so_13708826_13708925_2 . . . 67
3.1965 so_13708826_13708914_2 . . . 67
3.1966 so_24512191_24512804_0 . . . 67
3.1967 so_24512191_24512191_0 . . . 67
3.1968 so_19563891_19565257_1 . . . 67
3.1969 so_33126697_33127163_0 . . . 67
3.1970 so_23841022_23841234_3 . . . 67
3.1971 so_23841022_23841234_2 . . . 67
3.1972 so_33340718_33341499_0 . . . 67
3.1973 so_26648817_26649961_14 . . . 67
3.1974 so_26648817_26649961_11 . . . 67
3.1975 so_26648817_26649961_13 . . . 67
3.1976 so_24578654_24578654_0 . . . 67
3.1977 so_44613079_44613293_2 . . . 67
3.1978 so_44613079_44613293_3 . . . 67
3.1979 so_16782521_16782590_0 . . . 67
3.1980 so_24496795_24496795_0 . . . 67
3.1981 so_29312371_29312569_2 . . . 67
3.1982 so_29312371_29312569_0 . . . 67
3.1983 so_16300432_16302929_2 . . . 67
3.1984 so_16300432_16302929_0 . . . 67
3.1985 so_29230353_29236437_2 . . . 67
3.1986 so_29230353_29236437_0 . . . 67
3.1987 so_19069766_19069766_0 . . . 67
3.1988 so_18834855_18835188_0 . . . 67
3.1989 so_37655925_37655925_0 . . . 67
3.1990 so_37655925_37655925_2 . . . 67
3.1991 so_20788960_20789142_2 . . . 67
3.1992 so_20788960_20789054_0 . . . 67
3.1993 so_35886358_35886494_3 . . . 67
3.1994 so_28001512_28001552_1 . . . 67
3.1995 so_19176287_19177818_4 . . . 67
3.1996 so_5986668_5986678_0 . . . 67
3.1997 so_7074647_7074714_4 . . . 67
3.1998 so_7074647_7074714_2 . . . 67
3.1999 so_3877862_3877884_0 . . . 67
3.2000 so_3536372_3536498_6 . . . 67
3.2001 so_10066153_10066175_0 . . . 67
3.2002 so_2960307_2960663_0 . . . 67
3.2003 so_8353414_8353482_0 . . . 67
3.2004 so_4806510_4817981_2 . . . 67
3.2005 so_4114794_7378683_2 . . . 67
3.2006 so_43276872_43277929_1 . . . 67
3.2007 so_14782176_14782518_0 . . . 67
3.2008 so_30048154_30048403_0 . . . 67

3.2009 so_60653_60707_0 . . . 67
3.2010 so_55510_58804_0 . . . 67
3.2011 so_34131331_34131331_1 . . . 67
3.2012 so_18542804_18543786_1 . . . 67
3.2013 so_8362921_8363026_0 . . . 67
3.2014 so_24492190_24492190_0 . . . 67
3.2015 so_24492190_24493590_0 . . . 67
3.2016 so_22783166_22783340_6 . . . 67
3.2017 so_22783166_22783340_2 . . . 67
3.2018 so_22783166_22783340_3 . . . 67
3.2019 so_22783166_22783340_5 . . . 67
3.2020 so_44232434_44232513_2 . . . 67
3.2021 so_44232434_44232513_0 . . . 67
3.2022 so_33913670_33913670_0 . . . 67
3.2023 so_45524627_45524654_1 . . . 67
3.2024 so_29775826_29776416_0 . . . 67
3.2025 so_29775826_29776326_0 . . . 67
3.2026 so_3785808_3822635_1 . . . 67
3.2027 so_29109654_29109746_0 . . . 67
3.2028 so_44840066_44841232_0 . . . 67
3.2029 so_27197464_27197836_3 . . . 67
3.2030 so_27197464_27197464_0 . . . 67
3.2031 so_27197464_27197836_2 . . . 67
3.2032 so_27197464_27197836_4 . . . 67
3.2033 so_31045735_31045964_1 . . . 67
3.2034 so_18166500_18166662_0 . . . 67
3.2035 so_10256545_10256800_0 . . . 67
3.2036 so_38538855_38538983_0 . . . 67
3.2037 so_17443163_17443575_0 . . . 67
3.2038 so_25043717_25043996_0 . . . 67
3.2039 so_42206540_42208278_2 . . . 67
3.2040 so_22991549_22991760_1 . . . 67
3.2041 so_22991549_22991760_0 . . . 67
3.2042 so_24198158_24198194_0 . . . 67
3.2043 so_8750407_8750473_2 . . . 67
3.2044 so_8750407_8750473_0 . . . 67
3.2045 so_20857005_20857203_0 . . . 67
3.2046 so_13110522_13110647_0 . . . 67
3.2047 so_15094514_15094780_2 . . . 67
3.2048 so_15094514_15094780_0 . . . 67
3.2049 so_31383759_31384223_0 . . . 67
3.2050 so_31383759_31384223_1 . . . 67
3.2051 so_21964120_21964142_0 . . . 67
3.2052 so_1421671_1421780_0 . . . 67
3.2053 so_26701496_26702382_2 . . . 67
3.2054 so_26701496_26702382_0 . . . 67
3.2055 so_23734602_23734824_2 . . . 67
3.2056 so_23734602_23734824_0 . . . 67
3.2057 so_40107792_40108049_2 . . . 67
3.2058 so_19658609_19658739_2 . . . 67
3.2059 so_19658609_19658739_0 . . . 67

3.2060 so_6688118_6688543_0 . . . 67
3.2061 so_4377483_4377581_0 . . . 67
3.2062 so_14407044_14407138_1 . . . 67
3.2063 so_14407044_14407138_0 . . . 67
3.2064 so_29853235_29853544_2 . . . 67
3.2065 so_29853235_29853544_0 . . . 67
3.2066 so_25528245_25528459_0 . . . 67
3.2067 so_21771066_21771741_0 . . . 67
3.2068 so_30553957_30553971_0 . . . 67
3.2069 so_41596068_41596068_2 . . . 67
3.2070 so_37705923_37706327_0 . . . 67
3.2071 so_37705923_37706327_1 . . . 67
3.2072 so_15212261_15212447_0 . . . 67
3.2073 so_3951732_3951744_0 . . . 67
3.2074 so_2373859_2373895_0 . . . 67
3.2075 so_16373831_16374286_1 . . . 67
3.2076 so_3939810_3939853_0 . . . 67
3.2077 so_6516557_6516695_0 . . . 67
3.2078 so_23128045_23128045_4 . . . 67
3.2079 so_10011039_10011133_0 . . . 67
3.2080 so_39719557_39720975_1 . . . 67
3.2081 so_1819131_1825872_3 . . . 67
3.2082 so_8400791_8401267_1 . . . 67
3.2083 so_8400791_8401075_0 . . . 67
3.2084 so_34276373_34276374_5 . . . 67
3.2085 so_34276373_34276373_0 . . . 67
3.2086 so_34276373_34276374_6 . . . 67
3.2087 so_34276373_34276374_1 . . . 67
3.2088 so_34276373_34276374_3 . . . 67
3.2089 so_38215743_38215816_0 . . . 67
3.2090 so_3946641_3946694_1 . . . 67
3.2091 so_20865354_20865525_0 . . . 67
3.2092 so_18352385_18352801_4 . . . 67
3.2093 so_33441087_33441250_2 . . . 67
3.2094 so_19904461_19904759_0 . . . 67
3.2095 so_37774558_37794674_1 . . . 67
3.2096 so_37774558_37794674_2 . . . 67
3.2097 so_20862774_20862774_2 . . . 67
3.2098 so_20862774_20862774_6 . . . 67
3.2099 so_20862774_20862774_5 . . . 67
3.2100 so_25783480_25783562_2 . . . 67
3.2101 so_31468247_31468361_1 . . . 67
3.2102 so_31468247_31468361_0 . . . 67
3.2103 so_26175697_26175697_3 . . . 67
3.2104 so_26175697_26175697_1 . . . 67
3.2105 so_26175697_26175697_2 . . . 67
3.2106 so_26175697_26175697_0 . . . 67
3.2107 so_23763748_24067465_3 . . . 67
3.2108 so_32948874_32949863_0 . . . 67
3.2109 so_5945897_5945936_0 . . . 67
3.2110 so_30885449_30885670_3 . . . 67

3.2111 so_30885449_30885670_5 . . . 67
3.2112 so_24531186_24531338_2 . . . 67
3.2113 so_24531186_24531843_0 . . . 67
3.2114 so_24531186_24531655_4 . . . 67
3.2115 so_24531186_24531655_2 . . . 67
3.2116 so_24531186_24531655_3 . . . 67
3.2117 so_24531186_24531186_0 . . . 67
3.2118 so_8395848_8395972_0 . . . 67
3.2119 so_46379738_46382753_0 . . . 67
3.2120 so_46379738_46382753_2 . . . 67
3.2121 so_46379738_46382753_3 . . . 67
3.2122 so_46379738_46382753_4 . . . 67
3.2123 so_19035859_19035980_0 . . . 67
3.2124 so_32085442_32085766_0 . . . 68
3.2125 so_22117310_22117407_0 . . . 68
3.2126 so_38962226_38962644_0 . . . 68
3.2127 so_29312371_29312569_4 . . . 68
3.2128 so_29312371_29312569_3 . . . 68
3.2129 so_16300432_16302929_5 . . . 68
3.2130 so_3270917_3275101_0 . . . 68
3.2131 so_8353414_8353464_0 . . . 68
3.2132 so_4806510_4817981_1 . . . 68
3.2133 so_43276872_43277929_2 . . . 68
3.2134 so_14407044_14407138_2 . . . 68
3.2135 so_1255460_1255659_1 . . . 68
3.2136 so_39719557_39720975_2 . . . 68
3.2137 so_1819131_1825872_1 . . . 68
3.2138 so_8400791_8400791_8 . . . 68
3.2139 so_8400791_8400791_6 . . . 68
3.2140 so_20862774_20862936_0 . . . 68
3.2141 so_20862774_20862936_1 . . . 68
3.2142 so_20862774_20862936_2 . . . 68
3.2143 so_24552283_24552283_0 . . . 68
3.2144 so_24552283_24552283_2 . . . 68
3.2145 so_24552283_24552283_1 . . . 68
3.2146 so_18600034_18600034_0 . . . 68
3.2147 so_18600034_18601535_0 . . . 68
3.2148 so_18600034_18600034_1 . . . 68
3.2149 so_36157169_36158564_0 . . . 68
3.2150 so_7923624_7925027_1 . . . 68
3.2151 so_7923624_7923641_0 . . . 68
3.2152 so_23519823_23520635_2 . . . 68
3.2153 so_1271248_1271692_0 . . . 68
3.2154 so_13793355_13793460_0 . . . 68
3.2155 so_42785951_42786540_1 . . . 68
3.2156 so_20601358_20601358_1 . . . 68
3.2157 so_20601358_20601358_2 . . . 68
3.2158 so_34357636_34386075_0 . . . 68
3.2159 so_16300432_16302929_3 . . . 68
3.2160 so_16300432_16300433_4 . . . 68
3.2161 so_16300432_16300433_0 . . . 68

3.2162 so_23360196_23360649_0 . . . 68
3.2163 so_22680757_22681468_0 . . . 68
3.2164 so_15234572_15234667_0 . . . 68
3.2165 so_3872772_3872870_0 . . . 68
3.2166 so_26780795_26780878_0 . . . 68
3.2167 so_24656088_25845039_0 . . . 68
3.2168 so_12321401_12321401_0 . . . 68
3.2169 so_21482880_21483076_0 . . . 68
3.2170 so_29775826_29776416_1 . . . 68
3.2171 so_6317796_6317903_0 . . . 68
3.2172 so_31383759_31384223_2 . . . 68
3.2173 so_1421671_1421780_1 . . . 68
3.2174 so_26701496_26702382_3 . . . 68
3.2175 so_47569422_47570802_1 . . . 68
3.2176 so_47569422_47576570_0 . . . 68
3.2177 so_47569422_47569422_0 . . . 68
3.2178 so_14407044_14407138_3 . . . 68
3.2179 so_35757109_35757258_0 . . . 68
3.2180 so_22870454_22870510_0 . . . 68
3.2181 so_38554741_38554741_0 . . . 68
3.2182 so_3939810_3939853_1 . . . 68
3.2183 so_24112555_24112694_0 . . . 68
3.2184 so_24112555_24112694_3 . . . 68
3.2185 so_6234791_6234909_0 . . . 68
3.2186 so_33375027_33375635_0 . . . 68
3.2187 so_9975672_9976188_0 . . . 68
3.2188 so_40377546_40377626_1 . . . 68
3.2189 so_1239235_1239457_0 . . . 68
3.2190 so_38184127_38184127_0 . . . 68
3.2191 so_6372032_6373620_0 . . . 68
3.2192 so_4403002_4404720_0 . . . 68
3.2193 so_30885449_30885670_0 . . . 68
3.2194 so_30823696_30823823_0 . . . 68
3.2195 so_8395848_8395972_1 . . . 68
3.2196 so_18600034_18600034_2 . . . 69
3.2197 so_7923624_7925027_2 . . . 69
3.2198 so_47241504_47241504_0 . . . 69
3.2199 so_29775826_29776416_2 . . . 69
3.2200 so_24112555_24112694_2 . . . 69
3.2201 so_10667564_10667694_0 . . . 69
3.2202 so_14278006_14278575_0 . . . 69
3.2203 so_41869849_41870007_0 . . . 69
3.2204 so_23842433_23842564_1 . . . 69
3.2205 so_39539577_39539765_1 . . . 69
3.2206 so_43083856_43083976_0 . . . 69
3.2207 so_33820494_33820556_1 . . . 69
3.2208 so_30200893_30200992_0 . . . 69
3.2209 so_38130185_38132601_0 . . . 69
3.2210 so_40242063_40243538_0 . . . 69
3.2211 so_3044024_46068521_0 . . . 69
3.2212 so_9057613_9057644_4 . . . 69

3.2213 so_2204608_2204628_0 . . . 69
3.2214 so_4769229_4769305_1 . . . 69
3.2215 so_3785808_3822635_2 . . . 69
3.2216 so_27197464_27197551_0 . . . 69
3.2217 so_27197464_27197836_0 . . . 69
3.2218 so_27197464_27197836_1 . . . 69
3.2219 so_22343820_22344731_0 . . . 69
3.2220 so_26951007_26951007_0 . . . 69
3.2221 so_16010083_16010180_2 . . . 69
3.2222 so_24595372_24595372_0 . . . 69
3.2223 so_31443437_31443577_0 . . . 69
3.2224 so_31443437_31443437_1 . . . 69
3.2225 so_19492639_19493960_0 . . . 69
3.2226 so_41486703_41489471_3 . . . 69
3.2227 so_22476782_22478593_0 . . . 69
3.2228 so_10353501_10353501_0 . . . 70
3.2229 so_24612187_24612187_0 . . . 70
3.2230 so_41486703_41489471_5 . . . 70
3.2231 so_13586451_13586487_1 . . . 70
3.2232 so_16010083_16010180_3 . . . 70
3.2233 so_24631444_24631444_0 . . . 70
3.2234 so_17093164_17096858_0 . . . 70
3.2235 Storage duration . . . 70
3.2236 so_34586141_34595754_1 . . . 70
3.2237 so_24674609_24674609_0 . . . 70
3.2238 so_426737_426787_0 . . . 70
3.2239 so_35389789_35389789_6 . . . 70
3.2240 so_37640379_37640407_1 . . . 70
3.2241 so_24674609_24674609_1 . . . 70
3.2242 so_28790233_28790233_0 . . . 70
3.2243 so_36308071_36308300_1 . . . 71
3.2244 so_18944751_18944775_1 . . . 71
3.2245 so_15598023_15598100_1 . . . 71
3.2246 so_426737_426787_2 . . . 71
3.2247 so_18149191_18149441_2 . . . 71
3.2248 so_26972373_26972528_1 . . . 71
3.2249 so_24760041_24760066_1 . . . 71
3.2250 so_37796184_37796283_0 . . . 71
3.2251 so_34711966_34712225_0 . . . 71
3.2252 so_14034827_14034835_1 . . . 71
3.2253 so_1028443_1028511_0 . . . 71
3.2254 so_6205681_6205759_0 . . . 71
3.2255 so_41487900_41488409_0 . . . 71
3.2256 so_25926407_25926473_0 . . . 71
3.2257 so_14774152_14774430_1 . . . 71
3.2258 so_1588976_1589073_2 . . . 71
3.2259 so_15235526_15235526_2 . . . 71
3.2260 so_2252314_2252363_1 . . . 71
3.2261 so_8391058_8391983_4 . . . 71
3.2262 so_27149767_27149883_0 . . . 71
3.2263 so_27189541_27189594_2 . . . 71

3.2264 so_23128045_23128045_1 . . . 71
3.2265 so_430401_430513_1 . . . 71
3.2266 so_33441087_33441250_1 . . . 71
3.2267 so_15163574_15163758_0 . . . 71
3.2268 so_3047567_3047593_1 . . . 71
3.2269 so_33012680_33012777_2 . . . 71
3.2270 so_8171337_8171388_1 . . . 71
3.2271 so_6628908_6629002_3 . . . 71
3.2272 so_41869849_41870007_1 . . . 71
3.2273 so_23519823_23520635_1 . . . 71
3.2274 so_1315292_1315300_0 . . . 71
3.2275 so_8290132_8290523_0 . . . 71
3.2276 so_8290132_8290368_2 . . . 71
3.2277 so_8290132_8290368_1 . . . 71
3.2278 so_9795846_9796133_1 . . . 71
3.2279 so_44753653_44753852_0 . . . 71
3.2280 so_25926407_25926473_1 . . . 71
3.2281 so_430401_430513_2 . . . 71
3.2282 so_24678646_24678646_0 . . . 71
3.2283 so_34711966_34712225_1 . . . 71
3.2284 so_25926407_25926473_2 . . . 71
3.2285 so_14774152_14774430_2 . . . 71
3.2286 so_15235526_15235526_0 . . . 71
3.2287 so_8391058_8391983_5 . . . 71
3.2288 so_10816564_10817071_0 . . . 71
3.2289 so_25926407_25926473_3 . . . 71
3.2290 so_15235526_15235526_1 . . . 71
3.2291 so_15163574_15163758_1 . . . 71
3.2292 so_28454954_28455153_0 . . . 71
3.2293 so_17471525_17472621_1 . . . 71
3.2294 so_24701438_24701746_0 . . . 71
3.2295 so_24253584_24253610_0 . . . 71
3.2296 so_38130185_38132601_2 . . . 71
3.2297 so_9057613_9057644_5 . . . 71
3.2298 so_13110522_13111285_0 . . . 71
3.2299 so_28454954_28455153_1 . . . 71
3.2300 so_23555612_23558302_3 . . . 71
3.2301 so_47498215_47498351_0 . . . 71
3.2302 so_18409639_18409733_0 . . . 71
3.2303 so_426737_426787_4 . . . 71
3.2304 so_33024095_33024116_0 . . . 71
3.2305 so_28727439_28728510_0 . . . 71
3.2306 so_11163212_11163410_0 . . . 71
3.2307 so_15777133_15779613_6 . . . 71
3.2308 so_16265507_16265530_0 . . . 71
3.2309 so_19121820_19121880_0 . . . 71
3.2310 so_30380631_30380688_1 . . . 71
3.2311 so_18225064_18225160_1 . . . 71
3.2312 so_17356048_17356131_0 . . . 71
3.2313 so_30172355_30172469_3 . . . 71
3.2314 so_11637611_11637902_4 . . . 71

3.2315 so_2196327_2196495_3 . . . 71
3.2316 so_18599614_18599933_2 . . . 71
3.2317 so_8822229_8918505_4 . . . 71
3.2318 so_41373494_41374318_1 . . . 71
3.2319 so_6423210_6423346_0 . . . 71
3.2320 so_39340168_39342355_0 . . . 71
3.2321 so_19046608_19046754_0 . . . 71
3.2322 so_15777133_15779613_5 . . . 71
3.2323 so_37659063_37659207_0 . . . 71
3.2324 so_36816894_36817210_0 . . . 71
3.2325 so_16759733_16759922_1 . . . 71
3.2326 so_17356048_17356131_2 . . . 71
3.2327 so_27741698_27741897_0 . . . 71
3.2328 so_12326554_12373331_0 . . . 71
3.2329 so_7470913_7470913_0 . . . 71
3.2330 so_15454403_15454460_2 . . . 71
3.2331 so_118630_256351_1 . . . 71
3.2332 so_426737_426787_6 . . . 71
3.2333 so_42325978_42326111_12 . . . 71
3.2334 so_7964561_7964561_0 . . . 71
3.2335 so_21158278_21158309_2 . . . 71
3.2336 so_21158278_21158405_0 . . . 71
3.2337 so_38058106_38059018_2 . . . 71
3.2338 so_38058106_38059018_3 . . . 71
3.2339 so_426737_426787_8 . . . 72
3.2340 so_45689690_45690345_0 . . . 72
3.2341 so_42325978_42326111_13 . . . 72
3.2342 so_42325978_42326111_14 . . . 72
3.2343 so_34881205_34881205_1 . . . 72
3.2344 so_34881205_34881205_0 . . . 72
3.2345 so_36594569_36596739_0 . . . 72
3.2346 so_3742814_3742826_0 . . . 72
3.2347 so_2499895_2500002_0 . . . 72
3.2348 so_619467_619590_1 . . . 72
3.2349 so_13674332_13674353_1 . . . 72
3.2350 so_13674332_13674353_3 . . . 72
3.2351 so_13674332_13674353_2 . . . 72
3.2352 so_33704978_33705667_0 . . . 72
3.2353 so_38609960_38626627_1 . . . 72
3.2354 so_39078682_39078799_0 . . . 72
3.2355 so_24721713_24721755_0 . . . 72
3.2356 so_24721713_24721713_0 . . . 72
3.2357 so_24721713_24721776_0 . . . 72
3.2358 so_3821261_3821277_0 . . . 72
3.2359 so_42032760_42033459_0 . . . 72
3.2360 so_24683142_24684529_0 . . . 72
3.2361 so_31436620_31483849_3 . . . 72
3.2362 so_6496912_6497126_1 . . . 72
3.2363 so_6210921_6210992_0 . . . 72
3.2364 so_25717639_25717678_0 . . . 72
3.2365 so_35027861_35028046_0 . . . 72

3.2366 so_39961314_39961431_0 . . . 72
3.2367 so_19686769_19688690_1 . . . 72
3.2368 so_6065814_6065831_1 . . . 72
3.2369 so_6065814_6065831_0 . . . 72
3.2370 so_24843541_24843581_1 . . . 72
3.2371 so_24843541_24843541_0 . . . 72
3.2372 so_34531620_34531811_0 . . . 72
3.2373 so_46018409_46018409_0 . . . 72
3.2374 so_41385355_41388497_1 . . . 72
3.2375 so_33758838_33759192_0 . . . 72
3.2376 so_33758838_33759192_1 . . . 72
3.2377 so_3255671_3256565_0 . . . 72
3.2378 so_45481679_45482711_0 . . . 72
3.2379 so_5559525_5559525_1 . . . 72
3.2380 so_506518_506590_0 . . . 72
3.2381 so_14782176_14782518_2 . . . 72
3.2382 so_7058377_7058502_0 . . . 72
3.2383 so_33059531_33059591_0 . . . 72
3.2384 so_11033120_11033396_2 . . . 72
3.2385 so_11033120_11033396_0 . . . 72
3.2386 so_1259803_1259842_1 . . . 72
3.2387 so_15604411_15604573_2 . . . 72
3.2388 so_39872246_39873599_0 . . . 72
3.2389 so_24811030_24811030_0 . . . 72
3.2390 so_1719607_1719609_0 . . . 72
3.2391 so_24831435_24831435_0 . . . 72
3.2392 so_745850_751173_4 . . . 72
3.2393 so_31575140_31575140_1 . . . 72
3.2394 so_1087042_1087066_1 . . . 72
3.2395 so_1087042_1087066_2 . . . 72
3.2396 so_1087042_1087066_3 . . . 72
3.2397 so_1136844_1136844_3 . . . 72
3.2398 so_1136844_1136844_8 . . . 72
3.2399 so_1136844_1136844_5 . . . 72
3.2400 so_39137335_39138194_0 . . . 72
3.2401 so_38677277_38677532_0 . . . 72
3.2402 so_19022663_19023271_2 . . . 72
3.2403 so_7194127_7194137_1 . . . 72
3.2404 so_7194127_7194137_0 . . . 72
3.2405 so_45364539_45364781_0 . . . 72
3.2406 so_12038084_12038228_2 . . . 72
3.2407 so_4009463_4009509_0 . . . 72
3.2408 so_24683142_24684529_1 . . . 72
3.2409 so_31483064_31484575_0 . . . 72
3.2410 so_17571103_17573387_1 . . . 72
3.2411 so_31436620_31436956_0 . . . 72
3.2412 so_31436620_31483849_1 . . . 72
3.2413 so_6939989_6940763_0 . . . 72
3.2414 so_6973995_16510895_0 . . . 72
3.2415 so_28029980_28029980_0 . . . 72
3.2416 so_5216071_5216238_0 . . . 72

3.2417 so_11514185_11514185_0 . . . 73
3.2418 so_45481679_45482711_1 . . . 73
3.2419 so_33975641_33975641_0 . . . 73
3.2420 so_11033120_11033396_1 . . . 73
3.2421 so_35027861_35028046_1 . . . 73
3.2422 so_9456728_9459712_0 . . . 73
3.2423 so_45497684_46754122_0 . . . 73
3.2424 so_7194127_7194149_0 . . . 73
3.2425 so_24853959_24854152_0 . . . 73
3.2426 so_21644735_21645818_2 . . . 73
3.2427 so_20966681_20966744_1 . . . 73
3.2428 so_27259652_27259902_1 . . . 73
3.2429 so_19686769_19688690_6 . . . 73
3.2430 so_45149756_45150017_0 . . . 73
3.2431 so_24908550_24908550_0 . . . 73
3.2432 so_24908550_24908550_1 . . . 73
3.2433 so_33975641_33975641_1 . . . 73
3.2434 so_3821852_3822007_1 . . . 73
3.2435 so_33998748_33998748_0 . . . 73
3.2436 so_33998748_33999138_5 . . . 73
3.2437 so_24934981_24934981_0 . . . 73
3.2438 so_24959192_24959192_0 . . . 73
3.2439 so_24959192_24959192_1 . . . 73
3.2440 so_24959192_24959519_2 . . . 73
3.2441 so_19686769_19688690_7 . . . 73
3.2442 so_31436620_31483849_0 . . . 73
3.2443 so_24940403_24941118_0 . . . 73
3.2444 so_4191680_4191969_0 . . . 73
3.2445 so_5367674_5367826_0 . . . 73
3.2446 so_44161611_44167468_6 . . . 73
3.2447 so_44161611_44167468_5 . . . 73
3.2448 so_44161611_44167468_4 . . . 73
3.2449 so_5820530_5820752_0 . . . 73
3.2450 so_6562771_6562786_0 . . . 73
3.2451 so_19755033_19755953_1 . . . 73
3.2452 so_9570823_9570905_0 . . . 73
3.2453 so_441831_12513513_0 . . . 73
3.2454 so_25299005_25299017_2 . . . 73
3.2455 so_7419078_7419103_0 . . . 73
3.2456 so_16934982_16935018_1 . . . 73
3.2457 so_10218182_10218301_0 . . . 73
3.2458 so_19044809_19047503_0 . . . 73
3.2459 so_6890883_6890929_0 . . . 73
3.2460 so_328834_328909_2 . . . 73
3.2461 so_4480722_4481176_0 . . . 73
3.2462 so_25592317_25592331_0 . . . 73
3.2463 so_7626786_7626796_0 . . . 73
3.2464 so_45087706_45088068_0 . . . 73
3.2465 so_39992588_39992588_0 . . . 73
3.2466 so_8314739_8314757_0 . . . 73
3.2467 so_3821852_3822007_2 . . . 73

3.2468 so_9936011_9936077_0 . . . 73
3.2469 so_9936011_9936090_0 . . . 73
3.2470 so_12376461_16571948_0 . . . 73
3.2471 so_42032760_42033459_1 . . . 73
3.2472 so_8067744_8067779_0 . . . 73
3.2473 so_24433250_24433250_0 . . . 73
3.2474 so_16334231_16334279_1 . . . 73
3.2475 so_11115922_11115968_0 . . . 73
3.2476 so_24472841_24472876_0 . . . 73
3.2477 so_12129992_12130028_1 . . . 73
3.2478 so_6731331_6731484_1 . . . 73
3.2479 so_47354881_47355199_2 . . . 73
3.2480 so_7194127_7195226_0 . . . 73
3.2481 so_4418220_4418374_0 . . . 73
3.2482 so_14425188_14425206_1 . . . 73
3.2483 so_27133056_27133124_3 . . . 73
3.2484 so_8501740_8501794_0 . . . 73
3.2485 so_47090640_47090896_1 . . . 73
3.2486 so_4417934_4417964_0 . . . 73
3.2487 so_16598727_16598736_1 . . . 73
3.2488 so_5401062_5424199_0 . . . 73
3.2489 so_39339768_39339883_0 . . . 73
3.2490 so_39339768_39339840_1 . . . 73
3.2491 so_37000978_37001173_1 . . . 74
3.2492 so_45149756_45876962_1 . . . 74
3.2493 so_45149756_45149756_1 . . . 74
3.2494 so_45149756_45149756_2 . . . 74
3.2495 so_45149756_45876962_3 . . . 74
3.2496 so_45149756_45876962_11 . . . 74
3.2497 so_45149756_45876962_10 . . . 74
3.2498 so_45149756_45149756_4 . . . 74
3.2499 so_45149756_45787047_0 . . . 74
3.2500 so_45149756_45876962_0 . . . 74
3.2501 so_20569852_20570181_0 . . . 74
3.2502 so_11688162_11688331_1 . . . 74
3.2503 so_45689690_45690345_2 . . . 74
3.2504 so_47062419_47062647_0 . . . 74
3.2505 so_26666834_26668180_1 . . . 74
3.2506 so_26666834_26668180_0 . . . 74
3.2507 so_3391335_3392172_1 . . . 74
3.2508 so_20711463_20711544_0 . . . 74
3.2509 so_15604411_15604573_3 . . . 74
3.2510 so_41386084_41386205_0 . . . 74
3.2511 so_30694069_30694084_2 . . . 74
3.2512 so_30694069_30694084_1 . . . 74
3.2513 so_42032760_42033459_2 . . . 74
3.2514 so_25517664_25517738_0 . . . 74
3.2515 so_10662587_10664042_0 . . . 74
3.2516 so_44182049_44182938_2 . . . 74
3.2517 so_44182049_44182938_0 . . . 74
3.2518 so_44182049_44182938_1 . . . 74

3.2519 so_44182049_44182938_3 . . . 74
3.2520 so_44191920_44192442_1 . . . 74
3.2521 so_44191920_44192442_0 . . . 74
3.2522 so_47227977_47229011_1 . . . 74
3.2523 so_25255389_25255407_1 . . . 74
3.2524 so_20725030_20730923_2 . . . 74
3.2525 so_20725030_20730923_0 . . . 74
3.2526 so_12129992_12130028_2 . . . 74
3.2527 so_7194127_7195226_1 . . . 74
3.2528 so_8201327_8201344_0 . . . 74
3.2529 so_35744832_35745378_4 . . . 74
3.2530 so_43508771_43508853_0 . . . 74
3.2531 so_43508771_43508853_2 . . . 74
3.2532 so_43508771_43508853_1 . . . 74
3.2533 so_25152260_25152260_2 . . . 74
3.2534 so_25058081_25058081_0 . . . 74
3.2535 so_46123867_46124019_1 . . . 74
3.2536 so_30089415_30089452_0 . . . 74
3.2537 so_9114657_9114776_2 . . . 74
3.2538 so_42570752_42570876_2 . . . 74
3.2539 so_47227977_47229011_0 . . . 74
3.2540 so_8404598_8404676_1 . . . 74
3.2541 so_25174236_25174236_0 . . . 74
3.2542 so_25174236_25174236_2 . . . 74
3.2543 so_25216568_25216568_1 . . . 74
3.2544 so_5820530_5820752_2 . . . 75
3.2545 so_25228351_25228351_0 . . . 75
3.2546 so_11688162_11688331_0 . . . 75
3.2547 so_42128911_42129251_4 . . . 75
3.2548 so_42128911_42129251_7 . . . 75
3.2549 so_26666834_26668180_2 . . . 75
3.2550 so_13743727_13743727_1 . . . 75
3.2551 so_24398426_24398510_0 . . . 75
3.2552 so_15157591_15157689_0 . . . 75
3.2553 so_35744832_35745355_2 . . . 75
3.2554 so_26420348_26421176_2 . . . 75
3.2555 so_13743357_13743395_2 . . . 75
3.2556 so_18140317_18140390_0 . . . 75
3.2557 so_426737_426787_10 . . . 75
3.2558 so_11738895_11738915_1 . . . 75
3.2559 so_26354758_26355150_0 . . . 75
3.2560 so_13743727_13743727_0 . . . 75
3.2561 so_5719636_5730378_0 . . . 75
3.2562 so_11637611_11637902_5 . . . 75
3.2563 Object lifetime . . . 75
3.2564 so_18407076_18407244_3 . . . 75
3.2565 so_18407076_18407244_1 . . . 75
3.2566 so_18407076_18407244_2 . . . 75
3.2567 so_14444251_14444551_1 . . . 75
3.2568 so_14444251_14444576_0 . . . 75
3.2569 so_14444251_14444551_0 . . . 75

3.2570 so_46263697_46263908_0 . . . 75
3.2571 so_29150981_29151643_1 . . . 75
3.2572 so_29150981_29151144_3 . . . 75
3.2573 so_29150981_29151643_0 . . . 75
3.2574 so_29150981_29151020_0 . . . 75
3.2575 so_29150981_29151144_2 . . . 75
3.2576 so_25299995_25299995_0 . . . 75
3.2577 so_31669256_31669383_0 . . . 75
3.2578 so_37960088_37960952_1 . . . 75
3.2579 so_37960088_37960952_2 . . . 75
3.2580 so_24496795_24496820_0 . . . 75
3.2581 so_20409500_20409633_0 . . . 75
3.2582 so_42361656_42367505_0 . . . 75
3.2583 so_18279162_18279505_0 . . . 75
3.2584 so_35949849_35950168_3 . . . 75
3.2585 so_35949849_35950168_2 . . . 75
3.2586 so_35949849_35950168_4 . . . 75
3.2587 so_11373203_11996970_5 . . . 75
3.2588 so_41385355_41385385_1 . . . 75
3.2589 so_29132928_29145646_0 . . . 75
3.2590 so_37706858_37742564_1 . . . 75
3.2591 so_37706858_37742564_0 . . . 75
3.2592 so_37706858_37742564_2 . . . 75
3.2593 so_10401579_10403000_0 . . . 75
3.2594 so_37727096_37727096_1 . . . 75
3.2595 so_37727096_37727096_2 . . . 75
3.2596 so_37727096_37727096_0 . . . 75
3.2597 so_25300251_25300251_3 . . . 75
3.2598 so_25300251_25300251_4 . . . 75
3.2599 so_25300251_25300251_2 . . . 75
3.2600 so_25300251_25300251_0 . . . 75
3.2601 so_26171827_26171827_1 . . . 75
3.2602 so_26171827_26171827_0 . . . 75
3.2603 so_21101774_21147176_6 . . . 75
3.2604 so_21101774_21147176_7 . . . 75
3.2605 so_21101774_21147176_5 . . . 75
3.2606 so_4904762_4904884_0 . . . 75
3.2607 so_33649370_33651899_12 . . . 75
3.2608 so_33649370_33651899_13 . . . 75
3.2609 so_33649370_33651899_14 . . . 75
3.2610 so_45041045_45052333_3 . . . 75
3.2611 so_45041045_45052333_1 . . . 75
3.2612 so_45041045_45052333_2 . . . 75
3.2613 so_45041045_45052333_4 . . . 75
3.2614 so_45041045_45052333_5 . . . 75
3.2615 so_46387101_46387179_2 . . . 75
3.2616 so_46387101_46387179_1 . . . 75
3.2617 so_46387101_46387179_0 . . . 75
3.2618 so_34007190_34007608_2 . . . 75
3.2619 so_34007190_34007608_3 . . . 75
3.2620 so_14687396_14687642_1 . . . 75

3.2621 so_3063341_3063401_0 . . . 75
3.2622 so_3063341_3063401_1 . . . 75
3.2623 so_3063341_3063401_2 . . . 75
3.2624 so_11729603_11729729_0 . . . 75
3.2625 so_43887931_43888263_0 . . . 75
3.2626 so_19081578_19082038_5 . . . 75
3.2627 so_19081578_19082038_2 . . . 75
3.2628 so_19081578_19082038_0 . . . 75
3.2629 so_19081578_19082038_1 . . . 75
3.2630 so_19081578_19082038_3 . . . 75
3.2631 so_19081578_19082038_4 . . . 75
3.2632 so_6246365_6258431_0 . . . 75
3.2633 so_12612488_12615861_1 . . . 75
3.2634 so_12612488_12615861_2 . . . 75
3.2635 so_7086445_7086498_2 . . . 75
3.2636 so_30310299_30310501_2 . . . 75
3.2637 so_30310299_30315090_0 . . . 75
3.2638 so_30310299_30310428_0 . . . 75
3.2639 so_33514809_33515129_0 . . . 75
3.2640 so_46909105_46909166_4 . . . 75
3.2641 so_7526595_7526836_0 . . . 75
3.2642 so_28928590_28930230_2 . . . 75
3.2643 so_30466949_30468523_0 . . . 75
3.2644 so_15359654_15359823_0 . . . 75
3.2645 so_30315524_30315524_2 . . . 75
3.2646 so_30315524_30315524_3 . . . 75
3.2647 so_30315524_30315524_1 . . . 75
3.2648 so_8611468_8611468_0 . . . 75
3.2649 so_25697706_25697972_0 . . . 75
3.2650 so_5719636_5730378_3 . . . 75
3.2651 so_5719636_5730378_2 . . . 75
3.2652 so_5719636_5730378_1 . . . 75
3.2653 so_33706528_33706588_6 . . . 75
3.2654 so_33706528_33706588_5 . . . 75
3.2655 so_33706528_33706588_2 . . . 75
3.2656 so_33706528_33706588_1 . . . 75
3.2657 so_33706528_33706588_3 . . . 75
3.2658 so_33706528_33706588_0 . . . 75
3.2659 so_33706528_33706588_4 . . . 75
3.2660 so_24047340_24048582_2 . . . 75
3.2661 so_24047340_24048582_3 . . . 75
3.2662 so_2271046_2271687_1 . . . 75
3.2663 so_45628694_45628834_2 . . . 75
3.2664 so_45628694_45628834_3 . . . 75
3.2665 so_45628694_45628834_4 . . . 75
3.2666 so_4637774_4639568_0 . . . 75
3.2667 so_22808067_22808068_0 . . . 75
3.2668 so_20197636_20197637_0 . . . 75
3.2669 so_31483064_31484575_1 . . . 75
3.2670 so_31483064_31483328_0 . . . 75
3.2671 so_31483064_31484575_2 . . . 75

3.2672 so_33058717_33058919_3 . . . 75
3.2673 so_33058717_33058919_2 . . . 75
3.2674 so_33058717_33058919_4 . . . 75
3.2675 so_33058717_33058919_5 . . . 75
3.2676 so_30114397_30114590_0 . . . 75
3.2677 so_21806154_21806862_3 . . . 75
3.2678 so_46427709_46428390_0 . . . 75
3.2679 so_46427709_46428390_2 . . . 75
3.2680 so_46427709_46428390_3 . . . 75
3.2681 so_46427709_46428390_1 . . . 75
3.2682 so_15131140_15132284_0 . . . 75
3.2683 so_3291507_3291787_2 . . . 75
3.2684 so_3291507_3291787_3 . . . 75
3.2685 so_3291507_3291787_1 . . . 75
3.2686 so_11637611_11637902_2 . . . 75
3.2687 so_11637611_11637902_3 . . . 75
3.2688 so_23763748_34477904_0 . . . 75
3.2689 so_23763748_24067465_0 . . . 75
3.2690 so_35383453_35383576_2 . . . 75
3.2691 so_17619015_17620232_0 . . . 75
3.2692 so_37644977_37645073_0 . . . 75
3.2693 so_37644977_37645073_1 . . . 75
3.2694 so_30350127_30350754_0 . . . 75
3.2695 so_46838729_46860463_2 . . . 75
3.2696 so_46838729_46838940_2 . . . 75
3.2697 so_46838729_46838940_0 . . . 75
3.2698 so_46838729_46860463_0 . . . 75
3.2699 so_46838729_46860463_1 . . . 75
3.2700 so_46838729_46838940_1 . . . 75
3.2701 so_18659427_18664954_1 . . . 75
3.2702 so_18659427_18664954_0 . . . 75
3.2703 so_31360625_31360942_0 . . . 75
3.2704 so_25300251_25300251_6 . . . 75
3.2705 so_15938077_15938128_0 . . . 75
3.2706 so_7269099_24318220_0 . . . 75
3.2707 so_29150981_29151144_4 . . . 75
3.2708 so_24496795_24496820_1 . . . 75
3.2709 so_45041045_45052333_0 . . . 75
3.2710 so_11729603_11729729_5 . . . 75
3.2711 so_7086445_7086498_5 . . . 75
3.2712 so_24432603_24432847_0 . . . 75
3.2713 so_33706528_33706588_7 . . . 75
3.2714 so_14730534_14731285_0 . . . 75
3.2715 so_35383453_35383576_3 . . . 75
3.2716 so_31360625_31360942_2 . . . 75
3.2717 so_10277817_10277856_0 . . . 75
3.2718 so_39560093_39560247_1 . . . 75
3.2719 so_27552466_27552592_0 . . . 75
3.2720 so_1978709_24137969_0 . . . 75
3.2721 so_1978709_24137466_0 . . . 75
3.2722 so_1978709_24137427_0 . . . 75

3.2723 so_9971559_9971774_1 . . . 75
3.2724 so_9971559_9971608_0 . . . 75
3.2725 so_9971559_9971559_0 . . . 75
3.2726 so_12313405_12314089_0 . . . 75
3.2727 so_41385355_41385385_0 . . . 75
3.2728 so_25741239_25741356_0 . . . 75
3.2729 so_24137006_24137006_0 . . . 75
3.2730 so_6495632_6496004_1 . . . 75
3.2731 so_25599196_25599196_0 . . . 75
3.2732 so_16879988_16879995_0 . . . 75
3.2733 so_46186159_46186650_0 . . . 75
3.2734 so_18486596_18486654_0 . . . 75
3.2735 so_25757677_25757793_0 . . . 75
3.2736 so_9117358_9117429_1 . . . 75
3.2737 so_32735296_32735690_1 . . . 75
3.2738 so_15488099_15491548_1 . . . 75
3.2739 so_18077898_18078119_0 . . . 75
3.2740 so_21110059_21286446_7 . . . 75
3.2741 so_21110059_21272556_0 . . . 75
3.2742 so_21110059_21272556_1 . . . 75
3.2743 so_21110059_21272556_2 . . . 75
3.2744 so_21110059_21110059_0 . . . 75
3.2745 so_2474018_2474021_1 . . . 75
3.2746 so_18624449_18629436_1 . . . 75
3.2747 so_25697706_25697972_1 . . . 75
3.2748 so_46754725_46755237_1 . . . 75
3.2749 so_44261086_44262043_0 . . . 75
3.2750 so_22808067_22808068_1 . . . 75
3.2751 so_39763548_39771930_0 . . . 75
3.2752 so_39763548_39780208_2 . . . 75
3.2753 so_8511478_8511478_0 . . . 75
3.2754 so_20197636_20197637_1 . . . 75
3.2755 so_14727801_14728485_0 . . . 75
3.2756 so_31483064_31483328_1 . . . 75
3.2757 so_20541987_20542142_0 . . . 75
3.2758 so_9920973_9921320_0 . . . 75
3.2759 so_26315899_26315899_1 . . . 75
3.2760 so_26315899_26315899_0 . . . 75
3.2761 so_35389789_35395517_1 . . . 75
3.2762 so_35389789_35395517_2 . . . 75
3.2763 so_35383453_35383576_4 . . . 75
3.2764 so_26500905_26501139_0 . . . 75
3.2765 so_26500905_26501135_1 . . . 75
3.2766 so_17619015_17620232_1 . . . 75
3.2767 so_23980132_23980152_1 . . . 75
3.2768 so_27175952_27176107_1 . . . 75
3.2769 so_42997440_42997720_0 . . . 75
3.2770 so_14444251_14444551_2 . . . 76
3.2771 so_46263697_46263908_1 . . . 76
3.2772 so_46263697_46263908_2 . . . 76
3.2773 so_35125956_35134612_0 . . . 76

3.2774 so_29150981_29151643_2 . . . 76
3.2775 so_29150981_29151144_5 . . . 76
3.2776 so_24113066_24113446_0 . . . 76
3.2777 so_25636753_25636753_0 . . . 76
3.2778 so_25636753_25636753_2 . . . 76
3.2779 so_40403417_40403682_5 . . . 76
3.2780 so_30051662_30051877_0 . . . 76
3.2781 so_18475338_18475470_0 . . . 76
3.2782 so_37960088_37960952_3 . . . 76
3.2783 so_28727439_28728510_1 . . . 76
3.2784 so_28727439_28745226_0 . . . 76
3.2785 so_29525797_29528757_2 . . . 76
3.2786 so_10401579_10403000_1 . . . 76
3.2787 so_6495632_6496004_2 . . . 76
3.2788 so_34477383_34492547_4 . . . 76
3.2789 so_25708892_25709019_0 . . . 76
3.2790 so_26354758_26354900_0 . . . 76
3.2791 so_26354758_26355150_1 . . . 76
3.2792 so_14687396_14687642_2 . . . 76
3.2793 so_19081578_19082038_7 . . . 76
3.2794 so_19081578_19082038_6 . . . 76
3.2795 so_6246365_6258431_4 . . . 76
3.2796 so_6246365_6258431_1 . . . 76
3.2797 so_7086445_7086498_4 . . . 76
3.2798 so_30310299_30310501_1 . . . 76
3.2799 so_30310299_30310428_2 . . . 76
3.2800 so_20857005_20857203_1 . . . 76
3.2801 so_28928590_28929939_2 . . . 76
3.2802 so_25697706_25697972_2 . . . 76
3.2803 so_30691144_30691144_0 . . . 76
3.2804 so_39763548_39780208_1 . . . 76
3.2805 so_14730534_14731285_1 . . . 76
3.2806 so_46181083_46181083_1 . . . 76
3.2807 so_20984808_20989131_0 . . . 76
3.2808 so_22098834_22098970_4 . . . 76
3.2809 so_41678530_41678886_3 . . . 76
3.2810 so_41678530_41678886_2 . . . 76
3.2811 so_26074373_26074486_1 . . . 76
3.2812 so_26074373_26074986_0 . . . 76
3.2813 so_4379337_4379612_0 . . . 76
3.2814 so_4379337_4379612_1 . . . 76
3.2815 so_31360625_31360942_4 . . . 76
3.2816 so_14444251_14444466_0 . . . 77
3.2817 so_42137253_42137370_0 . . . 77
3.2818 so_4368361_4370147_0 . . . 77
3.2819 so_47120674_47120937_2 . . . 77
3.2820 so_47120674_47120937_0 . . . 77
3.2821 so_24496795_24496820_2 . . . 77
3.2822 so_2484389_2484878_0 . . . 77
3.2823 so_28727439_28728070_0 . . . 77
3.2824 so_33649370_33651899_5 . . . 77

3.2825 so_33649370_33651899_3 . . . 77
3.2826 so_33649370_33651899_7 . . . 77
3.2827 so_33649370_33651899_6 . . . 77
3.2828 so_45902829_45905157_1 . . . 77
3.2829 so_34477383_34492547_0 . . . 77
3.2830 so_34477383_34492547_2 . . . 77
3.2831 so_34477383_34492547_5 . . . 77
3.2832 so_25708892_25708892_2 . . . 77
3.2833 so_25708892_25708892_0 . . . 77
3.2834 so_37724668_37725173_0 . . . 77
3.2835 so_8595061_8595523_1 . . . 77
3.2836 so_43887931_43888263_2 . . . 77
3.2837 so_32608458_32608695_5 . . . 77
3.2838 so_32608458_32608695_3 . . . 77
3.2839 so_32608458_32608695_6 . . . 77
3.2840 so_32608458_32608695_7 . . . 77
3.2841 so_6246365_6258431_2 . . . 77
3.2842 so_4815689_4815754_0 . . . 77
3.2843 so_15359654_15359823_1 . . . 77
3.2844 so_40577808_40577869_1 . . . 77
3.2845 so_25697706_25697972_3 . . . 77
3.2846 so_36922180_36922424_1 . . . 77
3.2847 so_14730534_14731279_2 . . . 77
3.2848 so_14730534_14731279_4 . . . 77
3.2849 so_14730534_14731279_3 . . . 77
3.2850 so_14730534_14731279_0 . . . 77
3.2851 so_15131140_15132284_1 . . . 77
3.2852 so_23763748_34477904_7 . . . 77
3.2853 so_23763748_34477904_4 . . . 77
3.2854 so_23763748_34477904_8 . . . 77
3.2855 so_23763748_24067465_1 . . . 77
3.2856 so_23763748_34477904_6 . . . 77
3.2857 so_26758267_26767656_3 . . . 77
3.2858 so_26758267_26767656_2 . . . 77
3.2859 so_22098834_22098970_2 . . . 77
3.2860 so_22098834_22098970_0 . . . 77
3.2861 so_22098834_22098970_5 . . . 77
3.2862 so_17187771_17187771_0 . . . 77
3.2863 so_45880561_45880827_0 . . . 77
3.2864 so_30462113_30473350_3 . . . 77
3.2865 so_30462113_30473350_4 . . . 77
3.2866 so_30462113_30473350_1 . . . 77
3.2867 so_30462113_30473350_0 . . . 77
3.2868 so_30462113_30473350_2 . . . 77
3.2869 so_4715740_4715888_0 . . . 77
3.2870 so_30051662_30051877_1 . . . 77
3.2871 so_25736840_25738746_1 . . . 77
3.2872 so_12512961_12513118_0 . . . 77
3.2873 so_12512961_12513772_0 . . . 77
3.2874 so_29132928_29145646_1 . . . 77
3.2875 so_47473621_47475941_2 . . . 77

3.2876 so_47473621_47475941_3 . . . 77
3.2877 so_47473621_47475941_0 . . . 77
3.2878 so_47473621_47475941_1 . . . 77
3.2879 so_47473621_47475941_4 . . . 77
3.2880 so_21101774_21147176_0 . . . 77
3.2881 so_21101774_21147176_3 . . . 77
3.2882 so_21101774_21147176_4 . . . 77
3.2883 so_21101774_21147176_1 . . . 77
3.2884 so_21101774_21147176_2 . . . 77
3.2885 so_45041045_45052333_18 . . . 77
3.2886 so_45041045_45052333_19 . . . 77
3.2887 so_45041045_45052333_16 . . . 77
3.2888 so_45041045_45052333_20 . . . 77
3.2889 so_45041045_45052333_17 . . . 77
3.2890 so_46237554_46237554_0 . . . 77
3.2891 so_46237554_46237554_2 . . . 77
3.2892 so_29253100_29253567_1 . . . 77
3.2893 so_25757677_25757677_0 . . . 77
3.2894 so_9117358_9117429_0 . . . 77
3.2895 so_27003727_27049038_0 . . . 77
3.2896 so_33514809_33515129_1 . . . 77
3.2897 so_46014164_46015272_0 . . . 77
3.2898 so_46014164_46015752_2 . . . 77
3.2899 so_41094195_41906951_3 . . . 77
3.2900 so_41094195_41906951_1 . . . 77
3.2901 so_41094195_41906951_0 . . . 77
3.2902 so_41094195_41906951_2 . . . 77
3.2903 so_41094195_41906951_4 . . . 77
3.2904 so_42947064_42948254_2 . . . 77
3.2905 so_42947064_42948254_4 . . . 77
3.2906 so_42947064_42948254_1 . . . 77
3.2907 so_42947064_42948254_0 . . . 77
3.2908 so_25697706_25697706_0 . . . 77
3.2909 so_25697706_25697896_0 . . . 77
3.2910 so_36779096_36782474_0 . . . 77
3.2911 so_31361306_31361306_2 . . . 77
3.2912 so_31361306_31361306_3 . . . 77
3.2913 so_31361306_31361306_1 . . . 77
3.2914 so_31361306_31361306_4 . . . 77
3.2915 so_31361306_31361306_0 . . . 77
3.2916 so_22808067_22808068_2 . . . 77
3.2917 so_28790233_28790269_0 . . . 77
3.2918 so_25657435_25658750_0 . . . 77
3.2919 so_25657435_25658750_2 . . . 77
3.2920 so_25657435_25658750_3 . . . 77
3.2921 so_25657435_25658750_1 . . . 77
3.2922 so_41456172_41457116_2 . . . 77
3.2923 so_41456172_41457116_0 . . . 77
3.2924 so_41456172_41457116_1 . . . 77
3.2925 so_42598915_42599381_0 . . . 77
3.2926 so_11637611_11637902_6 . . . 77

3.2927 so_35389789_35395517_4 . . . 77
3.2928 so_35389789_35395517_7 . . . 77
3.2929 so_35389789_35395517_5 . . . 77
3.2930 so_35389789_35395517_6 . . . 77
3.2931 so_35389789_35395517_3 . . . 77
3.2932 so_46164958_46178544_0 . . . 77
3.2933 so_46164958_46178544_2 . . . 77
3.2934 so_41678530_41678530_1 . . . 77
3.2935 so_41678530_41678886_4 . . . 77
3.2936 so_32043314_32043636_1 . . . 77
3.2937 so_12908398_12911760_0 . . . 77
3.2938 so_42997440_43001282_3 . . . 77
3.2939 so_42997440_43001282_4 . . . 77
3.2940 so_42997440_43001282_0 . . . 77
3.2941 so_42997440_43001282_2 . . . 77
3.2942 so_42997440_43001282_1 . . . 77
3.2943 so_25736840_25736840_0 . . . 78
3.2944 so_29132928_29145646_2 . . . 78
3.2945 so_27205927_27206109_0 . . . 78
3.2946 so_18667318_18667318_0 . . . 78
3.2947 so_9345146_9345202_0 . . . 78
3.2948 so_42598915_42599381_1 . . . 78
3.2949 so_35389789_35395517_0 . . . 78
3.2950 so_26500905_26500905_0 . . . 78
3.2951 so_27175952_27176107_2 . . . 78
3.2952 so_25757677_25757677_1 . . . 78
3.2953 so_29883327_29883695_0 . . . 78
3.2954 so_20197636_20197637_2 . . . 78
3.2955 Types . . . 78
3.2956 so_4242186_4242216_0 . . . 78
3.2957 so_22206044_22206844_0 . . . 78
3.2958 so_5129312_5129407_1 . . . 78
3.2959 so_41189577_41190370_1 . . . 78
3.2960 so_29777492_39021760_2 . . . 78
3.2961 so_10953392_10953417_0 . . . 78
3.2962 so_45149756_45876962_9 . . . 78
3.2963 so_45149756_45876962_8 . . . 78
3.2964 so_45149756_45149756_3 . . . 78
3.2965 so_43521206_43549526_0 . . . 78
3.2966 so_43521206_43549526_1 . . . 78
3.2967 so_13642381_13642381_0 . . . 78
3.2968 so_32997185_32997605_0 . . . 78
3.2969 so_24034296_24034296_13 . . . 78
3.2970 so_24034296_24034296_14 . . . 78
3.2971 so_24034296_24034296_12 . . . 78
3.2972 so_24034296_24034296_11 . . . 78
3.2973 so_26171827_26171827_2 . . . 78
3.2974 so_38862435_38862457_0 . . . 78
3.2975 so_47498585_47499066_2 . . . 78
3.2976 so_47498585_47499066_1 . . . 78
3.2977 so_34879858_34880264_1 . . . 78

3.2978 so_34879858_34880264_4 . . . 78
3.2979 so_34879858_34880264_2 . . . 78
3.2980 so_21176193_21176193_5 . . . 78
3.2981 so_31636648_31654890_1 . . . 78
3.2982 so_39763548_39764671_2 . . . 78
3.2983 so_17625635_17625758_0 . . . 78
3.2984 so_1863069_1863093_1 . . . 78
3.2985 so_26998571_26999748_4 . . . 78
3.2986 so_30114397_30114921_5 . . . 78
3.2987 so_37373006_37373209_3 . . . 78
3.2988 so_7624714_7624942_0 . . . 78
3.2989 so_43551665_43551805_0 . . . 78
3.2990 so_13025214_13025534_0 . . . 78
3.2991 so_7536153_7536230_0 . . . 78
3.2992 so_7536153_7536245_0 . . . 78
3.2993 so_46625279_46629589_0 . . . 78
3.2994 so_653336_653780_0 . . . 78
3.2995 so_29777492_39021760_4 . . . 79
3.2996 so_44863382_44863678_0 . . . 79
3.2997 so_22812183_22812183_3 . . . 79
3.2998 so_39026871_39028696_9 . . . 79
3.2999 so_32997185_32997605_1 . . . 79
3.3000 so_26171827_30114008_0 . . . 79
3.3001 so_34879858_34880264_3 . . . 79
3.3002 so_31636648_31654890_0 . . . 79
3.3003 so_39832733_39832905_0 . . . 79
3.3004 so_39763548_39764671_3 . . . 79
3.3005 so_30114397_30114921_6 . . . 79
3.3006 so_7536153_7536230_1 . . . 79
3.3007 so_7536153_7536185_0 . . . 79
3.3008 so_32043314_32043636_0 . . . 79
3.3009 so_33380742_33398698_5 . . . 79
3.3010 so_33380742_33398698_4 . . . 79
3.3011 so_12773640_12773640_1 . . . 79
3.3012 so_12773640_12773640_0 . . . 79
3.3013 so_26861834_26862026_1 . . . 79
3.3014 so_26228577_26228577_0 . . . 79
3.3015 so_18475338_18475470_1 . . . 79
3.3016 so_47444081_47444081_0 . . . 79
3.3017 so_12893954_12894647_2 . . . 79
3.3018 so_25853450_25853450_2 . . . 79
3.3019 so_25853450_25853450_3 . . . 79
3.3020 so_11373203_11996970_4 . . . 79
3.3021 so_26171827_30114310_1 . . . 79
3.3022 so_1745487_1745633_0 . . . 79
3.3023 so_28207856_28208832_0 . . . 79
3.3024 so_40108184_40108184_1 . . . 79
3.3025 so_46738487_46743940_1 . . . 79
3.3026 so_26241397_26241397_0 . . . 79
3.3027 so_46735324_46736040_3 . . . 79
3.3028 so_34611181_34613124_0 . . . 79

3.3029 so_13819820_13820031_0 . . . 79
3.3030 so_26200639_26200639_0 . . . 79
3.3031 so_26200639_26201073_1 . . . 79
3.3032 so_1863069_1863093_2 . . . 79
3.3033 so_24032322_24032323_8 . . . 79
3.3034 so_24032322_24032323_9 . . . 79
3.3035 so_33869832_33869976_5 . . . 79
3.3036 so_33869832_33869976_6 . . . 79
3.3037 so_30395840_30396103_1 . . . 79
3.3038 so_5429855_5430329_1 . . . 79
3.3039 so_5429855_5430329_0 . . . 79
3.3040 so_18948966_18949137_2 . . . 79
3.3041 so_24106502_24106820_6 . . . 79
3.3042 so_24106502_24106820_7 . . . 79
3.3043 so_39278200_39278200_1 . . . 79
3.3044 so_30155802_30156439_0 . . . 79
3.3045 so_71416_685239_1 . . . 79
3.3046 so_34972172_34973290_6 . . . 79
3.3047 so_34972172_34973290_5 . . . 79
3.3048 so_27080061_27080417_0 . . . 79
3.3049 so_17467777_17468140_0 . . . 79
3.3050 so_44548242_44549084_1 . . . 79
3.3051 so_8836941_8837699_0 . . . 79
3.3052 so_27556840_27557375_0 . . . 79
3.3053 so_16597389_16597390_0 . . . 79
3.3054 so_16597389_16597390_3 . . . 79
3.3055 so_19840937_19841594_0 . . . 79
3.3056 so_22430877_22430899_0 . . . 79
3.3057 so_25853258_25854076_1 . . . 79
3.3058 so_47449598_47449750_0 . . . 79
3.3059 so_25967060_25967060_3 . . . 79
3.3060 so_25967060_25967060_2 . . . 79
3.3061 so_16573191_16573219_1 . . . 79
3.3062 so_26009080_26009080_1 . . . 79
3.3063 so_24478830_24480354_0 . . . 79
3.3064 so_27456027_27456027_0 . . . 79
3.3065 so_27117169_27117169_13 . . . 79
3.3066 so_27117169_27117390_0 . . . 79
3.3067 so_42365881_42365881_0 . . . 79
3.3068 so_12485025_12485165_1 . . . 79
3.3069 so_71416_685239_2 . . . 79
3.3070 so_25967060_25967495_0 . . . 79
3.3071 so_25967060_25967060_1 . . . 79
3.3072 so_7210286_7210958_0 . . . 79
3.3073 so_27229578_27229578_6 . . . 80
3.3074 so_14799690_14800797_0 . . . 80
3.3075 so_10359590_10359808_4 . . . 80
3.3076 so_22206044_22206844_1 . . . 80
3.3077 so_27080061_27080417_1 . . . 80
3.3078 so_29319869_29319975_1 . . . 80
3.3079 so_27226680_27226816_1 . . . 80

3.3080 so_25853258_25853661_1 . . . 80
3.3081 so_28883510_28883895_10 . . . 80
3.3082 so_14016993_14017039_2 . . . 80
3.3083 so_27949748_27949844_1 . . . 80
3.3084 so_3850040_3850092_2 . . . 80
3.3085 so_12754886_12755118_2 . . . 80
3.3086 so_13407737_13428150_0 . . . 80
3.3087 so_3852768_3852768_3 . . . 80
3.3088 so_15283523_15284276_2 . . . 80
3.3089 so_5442717_5442736_2 . . . 80
3.3090 so_32997185_32997605_3 . . . 80
3.3091 so_20788960_20789142_0 . . . 80
3.3092 so_18581443_18581492_0 . . . 80
3.3093 so_30096911_30097049_0 . . . 80
3.3094 so_3021333_3021396_0 . . . 80
3.3095 so_19526526_19526908_0 . . . 80
3.3096 so_36098055_36098055_0 . . . 80
3.3097 so_36098055_36098055_20 . . . 80
3.3098 so_32694114_32694707_1 . . . 80
3.3099 so_38251844_38252065_0 . . . 80
3.3100 so_38251844_38252103_0 . . . 80
3.3101 so_38251844_38252103_1 . . . 80
3.3102 so_21670459_21670459_1 . . . 80
3.3103 so_38193204_38193204_1 . . . 80
3.3104 so_25528245_25528459_3 . . . 80
3.3105 so_10212202_10212273_0 . . . 80
3.3106 so_146452_146589_0 . . . 80
3.3107 so_1628311_1628396_1 . . . 80
3.3108 so_39300653_43576047_2 . . . 80
3.3109 so_25542390_25543147_2 . . . 80
3.3110 so_34856936_34857037_3 . . . 80
3.3111 so_7536153_7536230_2 . . . 80
3.3112 so_28883510_28883895_9 . . . 80
3.3113 so_28883510_28883895_8 . . . 80
3.3114 so_28883510_28883895_6 . . . 80
3.3115 so_27167958_27168696_6 . . . 80
3.3116 so_45071876_45072047_4 . . . 80
3.3117 so_45071876_45072047_2 . . . 80
3.3118 so_45071876_45072047_3 . . . 80
3.3119 so_26648817_26649961_8 . . . 80
3.3120 so_26648817_26649961_9 . . . 80
3.3121 so_26648817_26649961_6 . . . 80
3.3122 so_16063123_16063123_1 . . . 80
3.3123 so_33813929_33813930_2 . . . 80
3.3124 so_27903336_27904091_11 . . . 80
3.3125 so_27903336_27904091_9 . . . 80
3.3126 so_27903336_27904091_12 . . . 80
3.3127 so_35356210_35356268_6 . . . 80
3.3128 so_35356210_35356268_7 . . . 80
3.3129 so_35356210_35356268_5 . . . 80
3.3130 so_27486581_27486581_6 . . . 80

3.3131 so_27486581_27486581_15 . . . 80
3.3132 so_27486581_27486581_5 . . . 80
3.3133 so_27486581_27486581_12 . . . 80
3.3134 so_27486581_27486581_14 . . . 80
3.3135 so_27486581_27486581_7 . . . 80
3.3136 so_24492190_24493590_1 . . . 80
3.3137 so_27496004_28115954_0 . . . 80
3.3138 so_21787488_21787700_7 . . . 80
3.3139 so_21787488_21787700_8 . . . 80
3.3140 so_21787488_21787700_5 . . . 80
3.3141 so_29255322_29255535_5 . . . 80
3.3142 so_29255322_29255535_4 . . . 80
3.3143 so_28332078_28332609_1 . . . 80
3.3144 so_38103578_38104223_9 . . . 80
3.3145 so_38103578_38104223_8 . . . 80
3.3146 so_38103578_38104223_7 . . . 80
3.3147 so_26529842_26530462_0 . . . 81
3.3148 so_28521188_28522481_2 . . . 81
3.3149 so_37323232_37369101_1 . . . 81
3.3150 so_37373006_37373209_2 . . . 81
3.3151 so_33869832_33869976_3 . . . 81
3.3152 so_41720176_41720882_0 . . . 81
3.3153 so_2462624_2462788_3 . . . 81
3.3154 so_34522504_34522520_0 . . . 81
3.3155 so_12381855_12381912_0 . . . 81
3.3156 so_13547368_13547368_2 . . . 81
3.3157 so_14256695_14257140_1 . . . 81
3.3158 so_26249221_26249221_1 . . . 81
3.3159 so_26249221_26249221_0 . . . 81
3.3160 so_5566245_5566314_1 . . . 81
3.3161 so_5566245_5566373_0 . . . 81
3.3162 so_5566245_5566410_0 . . . 81
3.3163 so_32029213_32029331_0 . . . 81
3.3164 so_46320222_46321012_0 . . . 81
3.3165 so_32785675_32785675_0 . . . 81
3.3166 so_19444580_19446224_1 . . . 81
3.3167 so_33569303_33569726_1 . . . 81
3.3168 so_17082841_17082931_0 . . . 81
3.3169 so_45506400_45507183_10 . . . 81
3.3170 so_24034296_24034296_7 . . . 81
3.3171 so_24034296_24034296_2 . . . 81
3.3172 so_24034296_24034296_8 . . . 81
3.3173 so_24034296_24034296_4 . . . 81
3.3174 so_24034296_24034296_3 . . . 81
3.3175 so_24034296_24034296_6 . . . 81
3.3176 so_24034296_24034296_9 . . . 81
3.3177 so_24034296_24034296_5 . . . 81
3.3178 so_17874122_17876141_0 . . . 81
3.3179 so_13122410_13122595_3 . . . 81
3.3180 so_13122410_13122595_1 . . . 81
3.3181 so_11138188_11139915_0 . . . 81

3.3182 so_11774877_11774958_0 . . . 81
3.3183 so_28207856_28208832_1 . . . 81
3.3184 so_4266771_4266820_0 . . . 81
3.3185 so_3618419_3618419_0 . . . 81
3.3186 so_28012943_28013000_0 . . . 81
3.3187 so_17097537_17097575_1 . . . 81
3.3188 so_4329777_4329777_1 . . . 81
3.3189 so_27684934_27685077_0 . . . 81
3.3190 so_27684934_27685077_2 . . . 81
3.3191 so_17453738_17453787_0 . . . 81
3.3192 so_26241397_26241397_2 . . . 81
3.3193 so_26241397_26241397_1 . . . 81
3.3194 so_3768363_3768363_1 . . . 81
3.3195 so_24079001_24079229_9 . . . 81
3.3196 so_24079001_24079229_7 . . . 81
3.3197 so_24079001_24079229_6 . . . 81
3.3198 so_24079001_24079229_4 . . . 81
3.3199 so_8888748_8888811_1 . . . 81
3.3200 so_10151834_10152170_1 . . . 81
3.3201 so_17438792_17438809_0 . . . 81
3.3202 so_4708444_4709195_0 . . . 81
3.3203 so_29338126_29338126_0 . . . 81
3.3204 so_21176193_21176278_0 . . . 81
3.3205 so_22270918_22271436_1 . . . 81
3.3206 so_22270918_22271436_8 . . . 81
3.3207 so_22270918_22271436_10 . . . 81
3.3208 so_22270918_22271436_2 . . . 81
3.3209 so_22270918_22271436_7 . . . 81
3.3210 so_22270918_22271436_9 . . . 81
3.3211 so_31551888_31552168_2 . . . 81
3.3212 so_14961871_14962321_0 . . . 81
3.3213 so_1592277_1593006_0 . . . 81
3.3214 so_3949471_3972350_0 . . . 81
3.3215 so_34560711_34560871_2 . . . 81
3.3216 so_34519334_34519365_1 . . . 81
3.3217 so_1229751_1551599_0 . . . 81
3.3218 so_34737737_34738084_0 . . . 81
3.3219 so_26200639_26200639_1 . . . 81
3.3220 so_3279220_3284563_1 . . . 81
3.3221 so_23469586_37782374_1 . . . 81
3.3222 so_19757653_19758067_0 . . . 81
3.3223 so_24032322_24032323_5 . . . 81
3.3224 so_24032322_24032323_4 . . . 81
3.3225 so_24032322_24032323_1 . . . 81
3.3226 so_24032322_24032323_6 . . . 81
3.3227 so_24032322_24032323_3 . . . 81
3.3228 so_24032322_24032323_2 . . . 81
3.3229 so_32856481_32856592_0 . . . 81
3.3230 so_32856481_32856568_0 . . . 81
3.3231 so_18042223_18042447_0 . . . 81
3.3232 so_37202036_37202108_1 . . . 81

3.3233 so_18833269_18833339_0 . . . 81
3.3234 so_45839129_45839230_0 . . . 81
3.3235 so_34132487_34137493_1 . . . 81
3.3236 so_3557720_3557752_0 . . . 81
3.3237 so_27464867_27464881_1 . . . 81
3.3238 so_25453665_25453786_0 . . . 81
3.3239 so_15911714_16006093_10 . . . 81
3.3240 so_15911714_16002781_3 . . . 81
3.3241 so_23749822_23749989_2 . . . 81
3.3242 so_32530475_32530759_2 . . . 81
3.3243 so_18971732_18971763_0 . . . 81
3.3244 so_14256695_14257140_2 . . . 81
3.3245 so_13412850_13412875_0 . . . 81
3.3246 so_13412850_13412875_1 . . . 81
3.3247 so_41455265_41455322_0 . . . 81
3.3248 so_33446588_33446796_0 . . . 81
3.3249 so_39683781_39683907_4 . . . 81
3.3250 so_39683781_39683907_5 . . . 81
3.3251 so_6491938_6492024_0 . . . 81
3.3252 so_589575_3618674_1 . . . 81
3.3253 so_589575_3618674_0 . . . 81
3.3254 so_589575_589586_0 . . . 81
3.3255 so_17834838_17835094_0 . . . 81
3.3256 so_17834838_28352310_2 . . . 81
3.3257 so_11774877_11774958_1 . . . 81
3.3258 so_34964200_34964461_1 . . . 81
3.3259 so_3414233_3414268_0 . . . 81
3.3260 so_12865949_12866041_0 . . . 81
3.3261 so_42232121_42232208_1 . . . 81
3.3262 so_42232121_42232208_3 . . . 81
3.3263 so_34540278_34540711_4 . . . 81
3.3264 so_22893712_22893724_0 . . . 81
3.3265 so_4329777_4329777_0 . . . 81
3.3266 so_2432111_2432149_1 . . . 81
3.3267 so_31527219_31527797_0 . . . 81
3.3268 so_23554024_23554066_0 . . . 81
3.3269 so_5381882_30322474_0 . . . 81
3.3270 so_5381882_30322474_2 . . . 81
3.3271 so_23454557_23454711_1 . . . 81
3.3272 so_23454557_23454711_0 . . . 81
3.3273 so_271076_271107_1 . . . 81
3.3274 so_271076_3618662_0 . . . 81
3.3275 so_271076_3618662_1 . . . 81
3.3276 so_271076_271107_0 . . . 81
3.3277 so_11611467_11617699_2 . . . 81
3.3278 so_3618011_3618635_0 . . . 81
3.3279 so_3618011_3618635_1 . . . 81
3.3280 so_33571554_33573507_1 . . . 81
3.3281 so_21265462_21267714_2 . . . 81
3.3282 so_47067006_47068111_2 . . . 81
3.3283 so_7456902_7456932_0 . . . 81

3.3284 so_8674743_8674753_0 . . . 81
3.3285 so_22493879_22493995_0 . . . 81
3.3286 so_32857716_32859374_0 . . . 81
3.3287 so_18833269_18833339_2 . . . 81
3.3288 so_18833269_18833339_1 . . . 81
3.3289 so_26270470_26270470_0 . . . 81
3.3290 so_4329277_4329376_0 . . . 81
3.3291 so_4329277_4329376_1 . . . 81
3.3292 so_13446917_13446924_1 . . . 81
3.3293 so_21080532_21080853_2 . . . 81
3.3294 so_22204257_22204503_0 . . . 81
3.3295 so_15911714_16006093_8 . . . 81
3.3296 so_15911714_16002781_0 . . . 81
3.3297 so_15911714_16006093_5 . . . 81
3.3298 so_23373742_23375539_7 . . . 81
3.3299 so_3029412_3029439_0 . . . 81
3.3300 so_33380742_33398698_3 . . . 81
3.3301 so_33673798_33673798_2 . . . 81
3.3302 so_32700032_32700032_5 . . . 81
3.3303 so_33569303_33569726_2 . . . 81
3.3304 so_589575_589586_1 . . . 81
3.3305 so_3414233_3414268_1 . . . 81
3.3306 so_11985774_12338977_6 . . . 81
3.3307 so_23020323_23020323_1 . . . 81
3.3308 so_34540278_34540711_5 . . . 81
3.3309 so_28012943_28013000_1 . . . 81
3.3310 so_41276869_41276995_1 . . . 81
3.3311 so_34611181_34613124_1 . . . 81
3.3312 so_11224210_11224210_2 . . . 81
3.3313 so_11224210_11224210_4 . . . 81
3.3314 so_20407747_20407779_0 . . . 81
3.3315 so_31527219_31527797_1 . . . 81
3.3316 so_31551888_31552168_1 . . . 81
3.3317 so_36182654_36183315_3 . . . 81
3.3318 so_8366290_8366323_1 . . . 81
3.3319 so_11611467_11617699_3 . . . 81
3.3320 so_19471235_19471288_1 . . . 81
3.3321 so_34986335_34986391_3 . . . 81
3.3322 so_34148992_34151014_0 . . . 81
3.3323 so_34148992_34153294_0 . . . 81
3.3324 so_3618011_3618139_11 . . . 81
3.3325 so_11372044_11372091_0 . . . 81
3.3326 so_25940635_25941170_0 . . . 81
3.3327 so_16616299_16616334_0 . . . 81
3.3328 so_16616299_16616334_1 . . . 81
3.3329 so_13150449_13150449_0 . . . 81
3.3330 so_13150449_13150449_1 . . . 81
3.3331 so_13169451_13169473_1 . . . 81
3.3332 so_13169451_13169473_0 . . . 81
3.3333 so_27465728_27465758_0 . . . 81
3.3334 so_27465728_27465746_0 . . . 81

3.3335 so_21080532_21080853_3 . . . 81
3.3336 so_25453665_25453786_1 . . . 81
3.3337 so_25453665_25453786_2 . . . 81
3.3338 so_31332380_31332657_0 . . . 82
3.3339 so_32477249_32479159_4 . . . 82
3.3340 so_675976_676008_1 . . . 82
3.3341 so_3029412_3029439_1 . . . 82
3.3342 so_6842880_6843474_0 . . . 82
3.3343 so_6842880_6843474_2 . . . 82
3.3344 so_9474699_9474799_1 . . . 82
3.3345 so_21421998_21422105_0 . . . 82
3.3346 so_16188263_16188846_2 . . . 82
3.3347 so_16188263_16188846_3 . . . 82
3.3348 so_9052088_9053207_2 . . . 82
3.3349 so_17398959_17437214_1 . . . 82
3.3350 so_17398959_17437214_2 . . . 82
3.3351 so_5458204_5458251_1 . . . 82
3.3352 so_17176221_17176239_2 . . . 82
3.3353 so_17176221_17176239_3 . . . 82
3.3354 so_18195715_18195715_0 . . . 82
3.3355 so_19205125_19205206_1 . . . 82
3.3356 so_28779803_28779876_0 . . . 82
3.3357 so_28779803_28779876_1 . . . 82
3.3358 so_2151305_2151321_2 . . . 82
3.3359 so_2151305_2151335_0 . . . 82
3.3360 so_2151305_2151335_1 . . . 82
3.3361 so_9024826_9024826_1 . . . 82
3.3362 so_41276869_41277125_1 . . . 82
3.3363 so_41276869_41277125_2 . . . 82
3.3364 so_33732489_33732772_0 . . . 82
3.3365 so_11224210_11224210_3 . . . 82
3.3366 so_11224210_11224210_5 . . . 82
3.3367 so_12889237_12889697_3 . . . 82
3.3368 so_14065337_14088595_0 . . . 82
3.3369 so_1269019_1269049_0 . . . 82
3.3370 so_1269019_1269049_3 . . . 82
3.3371 so_1269019_1269049_2 . . . 82
3.3372 so_25921786_25921843_1 . . . 82
3.3373 so_25921786_25921843_0 . . . 82
3.3374 so_23166648_23191394_1 . . . 82
3.3375 so_27750818_27750835_0 . . . 82
3.3376 so_34858829_34859003_0 . . . 82
3.3377 so_34858829_34859199_1 . . . 82
3.3378 so_44477167_44477276_0 . . . 82
3.3379 so_12764629_12764649_1 . . . 82
3.3380 so_12764629_12764649_0 . . . 82
3.3381 so_21689626_21689662_0 . . . 82
3.3382 so_9897698_9897825_1 . . . 82
3.3383 so_9897698_9897825_0 . . . 82
3.3384 so_9897698_9897825_2 . . . 82
3.3385 so_35138864_35138909_3 . . . 82

3.3386 so_31702505_31702712_5 . . . 82
3.3387 so_31702505_31702712_6 . . . 82
3.3388 so_12275248_12275310_2 . . . 82
3.3389 so_12275248_12275269_0 . . . 82
3.3390 so_12275248_12275310_1 . . . 82
3.3391 so_20570125_20570165_0 . . . 82
3.3392 so_8208023_8209275_1 . . . 82
3.3393 so_24296571_24296605_5 . . . 82
3.3394 so_43715070_43715191_1 . . . 82
3.3395 so_6440812_6440983_0 . . . 82
3.3396 so_2395514_2395531_0 . . . 82
3.3397 so_15279080_15279157_0 . . . 82
3.3398 so_47444081_47444768_0 . . . 82
3.3399 so_27109701_27110545_6 . . . 82
3.3400 so_20958262_20958364_0 . . . 82
3.3401 so_26333381_26333381_0 . . . 82
3.3402 so_7232710_7235204_0 . . . 82
3.3403 so_20552491_20552491_0 . . . 82
3.3404 so_13710157_13710222_0 . . . 82
3.3405 so_13710157_13710198_0 . . . 82
3.3406 so_12798395_12798436_0 . . . 82
3.3407 so_39548465_39548869_0 . . . 82
3.3408 so_39548465_39548465_1 . . . 82
3.3409 so_38770680_38771118_0 . . . 82
3.3410 so_3257263_3278218_0 . . . 82
3.3411 so_11107608_11107667_0 . . . 82
3.3412 so_32856481_32856568_5 . . . 82
3.3413 so_32856481_32856650_0 . . . 82
3.3414 so_32856481_32856592_2 . . . 82
3.3415 so_30099819_30099819_0 . . . 82
3.3416 so_23580753_23580773_1 . . . 82
3.3417 so_23580753_23580808_0 . . . 82
3.3418 so_12789376_12789376_0 . . . 82
3.3419 so_25453665_25453786_3 . . . 82
3.3420 so_34122434_34122776_1 . . . 82
3.3421 so_4518951_4519145_0 . . . 82
3.3422 so_4518951_4519138_0 . . . 82
3.3423 so_4518951_4519145_1 . . . 82
3.3424 so_33380742_33398698_7 . . . 82
3.3425 so_14388855_14388886_0 . . . 82
3.3426 so_9498057_9498127_0 . . . 82
3.3427 so_25308761_25308881_2 . . . 82
3.3428 so_25308761_25308909_11 . . . 82
3.3429 so_25308761_25308909_10 . . . 82
3.3430 so_25308761_25308881_0 . . . 82
3.3431 so_28207856_28208832_3 . . . 82
3.3432 so_33981801_33982104_2 . . . 82
3.3433 so_33981801_33982104_1 . . . 82
3.3434 so_17501942_17503313_1 . . . 82
3.3435 so_18984934_18985007_1 . . . 82
3.3436 so_18984934_18985066_3 . . . 82

3.3437 so_18984934_18985085_1 . . . 82
3.3438 so_18984934_18985007_0 . . . 82
3.3439 so_3845353_3845403_1 . . . 82
3.3440 so_3845353_3845403_0 . . . 82
3.3441 so_17710421_17710673_0 . . . 82
3.3442 so_23268357_23268553_2 . . . 82
3.3443 so_23268357_23268553_0 . . . 82
3.3444 so_13254873_13254998_2 . . . 82
3.3445 so_4879045_4879045_2 . . . 82
3.3446 so_4879045_4879045_0 . . . 82
3.3447 so_34612344_34612470_0 . . . 82
3.3448 so_4276207_4276293_0 . . . 82
3.3449 so_26143976_26144237_2 . . . 82
3.3450 so_23874077_23874224_0 . . . 82
3.3451 so_4279264_4279901_1 . . . 82
3.3452 so_4279264_4279901_0 . . . 82
3.3453 so_18042223_18042273_1 . . . 82
3.3454 so_44826175_44828363_0 . . . 82
3.3455 so_44826175_44828363_1 . . . 82
3.3456 so_25453665_25453786_4 . . . 82
3.3457 so_17334126_17334170_0 . . . 82
3.3458 so_33380742_33380798_0 . . . 82
3.3459 so_33380742_33398698_6 . . . 82
3.3460 so_9656941_9657064_1 . . . 82
3.3461 so_14388855_14388887_2 . . . 82
3.3462 so_30254582_30869374_2 . . . 82
3.3463 so_12125650_12125836_4 . . . 82
3.3464 so_12125650_12125836_2 . . . 82
3.3465 so_12125650_12125836_1 . . . 82
3.3466 so_24708594_24708787_0 . . . 82
3.3467 so_24769567_24769567_0 . . . 82
3.3468 so_11138188_11138189_1 . . . 82
3.3469 so_11138188_11138189_0 . . . 82
3.3470 so_11138188_11139915_1 . . . 82
3.3471 so_13171054_13171148_1 . . . 82
3.3472 so_38466124_38467507_8 . . . 82
3.3473 so_38466124_38467507_7 . . . 82
3.3474 so_809227_852861_0 . . . 82
3.3475 so_606728_606889_2 . . . 82
3.3476 so_12655875_12656129_1 . . . 82
3.3477 so_12655875_12656129_0 . . . 82
3.3478 so_34611181_34613124_3 . . . 82
3.3479 so_34611181_34613124_2 . . . 82
3.3480 so_8288502_8288565_0 . . . 82
3.3481 so_31527219_31527797_2 . . . 82
3.3482 so_14676574_14702711_0 . . . 82
3.3483 so_44618373_44618474_1 . . . 82
3.3484 so_27507361_27507361_2 . . . 82
3.3485 so_27507361_27508570_7 . . . 82
3.3486 so_26143976_26144079_1 . . . 82
3.3487 so_34862241_34863695_1 . . . 82

3.3488 so_34862241_34863695_0 . . . 82
3.3489 so_21483564_21483704_0 . . . 82
3.3490 so_40607013_40607125_1 . . . 82
3.3491 so_40607013_40607125_2 . . . 82
3.3492 so_12695943_12696029_0 . . . 82
3.3493 so_12695943_12696029_1 . . . 82
3.3494 so_12695943_12696029_5 . . . 82
3.3495 so_13150449_13208789_3 . . . 82
3.3496 so_13150449_13208789_4 . . . 82
3.3497 so_27337141_27337319_0 . . . 82
3.3498 so_5654159_5654210_0 . . . 82
3.3499 so_8208023_8209275_2 . . . 82
3.3500 so_8208023_8209275_3 . . . 82
3.3501 so_45839129_45839230_2 . . . 82
3.3502 so_45839129_45839230_1 . . . 82
3.3503 so_29394518_29394720_1 . . . 82
3.3504 so_29394518_29394720_0 . . . 82
3.3505 so_22204257_22204503_1 . . . 82
3.3506 so_15911714_16002781_2 . . . 82
3.3507 so_15911714_16006093_9 . . . 82
3.3508 so_28935246_29875223_0 . . . 82
3.3509 so_28935246_29875223_1 . . . 82
3.3510 so_25302480_25302649_10 . . . 82
3.3511 so_22019261_22019445_5 . . . 82
3.3512 so_24406000_24406076_4 . . . 82
3.3513 so_14221612_14221894_0 . . . 82
3.3514 so_16590715_16590715_0 . . . 82
3.3515 so_30020005_30020407_2 . . . 82
3.3516 so_34898876_34899757_1 . . . 82
3.3517 so_29510356_29510356_0 . . . 82
3.3518 so_28045787_28046896_1 . . . 82
3.3519 so_11774877_11774958_3 . . . 82
3.3520 so_40307531_40307869_1 . . . 82
3.3521 so_44211841_44211915_1 . . . 82
3.3522 so_21706877_21707185_0 . . . 82
3.3523 so_2386772_2386808_0 . . . 82
3.3524 so_2386772_2386808_1 . . . 82
3.3525 so_26448637_26448992_0 . . . 82
3.3526 so_26448637_26448637_0 . . . 82
3.3527 so_3589663_3589663_1 . . . 82
3.3528 so_47128825_47129827_0 . . . 82
3.3529 so_20542655_20542831_0 . . . 82
3.3530 so_29904728_29904728_0 . . . 82
3.3531 so_31527219_31527797_3 . . . 82
3.3532 so_7580352_7580541_1 . . . 82
3.3533 so_20180969_20181524_1 . . . 82
3.3534 so_20180969_20181524_0 . . . 82
3.3535 so_27333887_27333940_0 . . . 82
3.3536 so_45481484_45482138_1 . . . 82
3.3537 so_20849674_20849769_0 . . . 82
3.3538 so_10771260_10771283_0 . . . 82

3.3539 so_9316476_9316506_0 . . . 82
3.3540 so_39300653_43576047_3 . . . 82
3.3541 so_27571156_27571696_1 . . . 82
3.3542 so_10775361_10775667_1 . . . 82
3.3543 so_26143976_26144079_2 . . . 82
3.3544 so_24157094_24157568_0 . . . 82
3.3545 so_38652155_38652313_0 . . . 82
3.3546 so_33363994_33364183_1 . . . 82
3.3547 so_14203654_14205392_2 . . . 82
3.3548 so_25524355_25524458_0 . . . 82
3.3549 so_6941733_6942028_0 . . . 82
3.3550 so_1816552_1816552_0 . . . 82
3.3551 so_14773142_14773502_0 . . . 82
3.3552 so_1610030_1610148_2 . . . 82
3.3553 so_18702195_18702231_0 . . . 82
3.3554 so_18822242_18823082_0 . . . 82
3.3555 so_17530279_17530324_2 . . . 82
3.3556 so_35987493_35987910_0 . . . 82
3.3557 so_35987493_35987890_2 . . . 82
3.3558 so_19482671_19482858_0 . . . 82
3.3559 so_39300653_43576047_4 . . . 82
3.3560 so_29319869_29319975_2 . . . 82
3.3561 so_25853258_25853258_0 . . . 82
3.3562 so_32867939_32868057_2 . . . 82
3.3563 so_21178903_21179084_1 . . . 82
3.3564 so_14092284_14092302_5 . . . 82
3.3565 so_2143906_2143995_0 . . . 82
3.3566 so_39946420_39946420_1 . . . 82
3.3567 so_3383090_3383099_1 . . . 82
3.3568 so_37127988_37128054_0 . . . 82
3.3569 so_17142130_17147477_0 . . . 83
3.3570 so_34540278_34540711_1 . . . 83
3.3571 so_41290827_41290827_1 . . . 83
3.3572 so_23716749_23717265_7 . . . 83
3.3573 so_23716749_23717265_9 . . . 83
3.3574 so_23716749_23717265_3 . . . 83
3.3575 so_23716749_23717265_10 . . . 83
3.3576 so_23716749_23717265_2 . . . 83
3.3577 so_23716749_23717265_0 . . . 83
3.3578 so_23716749_23717265_4 . . . 83
3.3579 so_23716749_23717265_8 . . . 83
3.3580 so_32842429_32845696_0 . . . 83
3.3581 so_28119793_28119888_0 . . . 83
3.3582 so_29319869_29319975_3 . . . 83
3.3583 so_29319869_29319975_4 . . . 83
3.3584 so_24867294_24868095_0 . . . 83
3.3585 so_33812998_33814842_2 . . . 83
3.3586 so_12317661_12317694_1 . . . 83
3.3587 so_29720288_29720461_1 . . . 83
3.3588 so_26626663_26627168_1 . . . 83
3.3589 so_9234217_9234299_0 . . . 83

3.3590 so_6998750_6999394_1 . . . 83
3.3591 so_27229578_27229664_1 . . . 83
3.3592 so_41848385_41849966_0 . . . 83
3.3593 so_8336220_8336232_1 . . . 83
3.3594 so_26917194_26917282_2 . . . 83
3.3595 so_3056174_3056631_1 . . . 83
3.3596 so_11647220_11647310_0 . . . 83
3.3597 so_11373203_11996970_6 . . . 83
3.3598 so_29277949_29278260_1 . . . 83
3.3599 so_16607094_16608475_1 . . . 83
3.3600 so_11871299_11871465_0 . . . 83
3.3601 so_37981617_37981656_2 . . . 83
3.3602 so_32546573_32546666_0 . . . 83
3.3603 so_496226_496381_0 . . . 83
3.3604 so_32100245_32100749_0 . . . 83
3.3605 so_32100245_32100749_1 . . . 83
3.3606 so_15547305_15547408_1 . . . 83
3.3607 so_34540278_34540711_3 . . . 83
3.3608 so_34540278_34540711_2 . . . 83
3.3609 so_14272141_14285078_0 . . . 83
3.3610 so_39477443_39478189_0 . . . 83
3.3611 so_31842836_31843306_2 . . . 83
3.3612 so_42492332_42493240_2 . . . 83
3.3613 so_20407747_20407956_4 . . . 83
3.3614 so_20407747_20407956_3 . . . 83
3.3615 so_26290770_26290930_1 . . . 83
3.3616 so_988158_988220_2 . . . 83
3.3617 so_28521188_28522481_3 . . . 83
3.3618 so_25524668_25525214_1 . . . 83
3.3619 so_46425250_46426307_1 . . . 83
3.3620 so_34737737_34748943_0 . . . 83
3.3621 so_21948280_21948531_0 . . . 83
3.3622 so_24398426_24398510_1 . . . 83
3.3623 so_7762929_7762964_4 . . . 83
3.3624 so_15217572_15221498_1 . . . 83
3.3625 so_20105329_20105367_2 . . . 83
3.3626 so_36713621_36716803_0 . . . 83
3.3627 so_36713621_36714381_0 . . . 83
3.3628 so_33869832_33869976_1 . . . 83
3.3629 so_39860732_39861534_4 . . . 83
3.3630 so_26420348_26421176_0 . . . 83
3.3631 so_27754055_27754244_3 . . . 83
3.3632 so_32942219_32942595_0 . . . 83
3.3633 so_32942219_32942869_0 . . . 83
3.3634 so_32738518_32739887_2 . . . 83
3.3635 so_23373742_23373833_0 . . . 84
3.3636 so_36790872_36790985_1 . . . 84
3.3637 so_24646084_24646191_0 . . . 84
3.3638 so_26917194_26917282_3 . . . 84
3.3639 so_16554788_16554829_1 . . . 84
3.3640 so_1863069_1863093_3 . . . 84

3.3641 so_17426476_17426570_1 . . . 84
3.3642 so_17426476_17426570_0 . . . 84
3.3643 so_16100069_16100415_2 . . . 84
3.3644 so_32106803_32106942_1 . . . 84
3.3645 so_36098055_36098055_19 . . . 84
3.3646 so_8952593_8952728_0 . . . 84
3.3647 so_38851469_38853231_0 . . . 84
3.3648 so_18586189_18586708_1 . . . 84
3.3649 so_18102156_18285077_0 . . . 84
3.3650 so_38799924_38799957_0 . . . 84
3.3651 so_38799924_38799954_0 . . . 84
3.3652 so_15413037_15413389_4 . . . 84
3.3653 so_15413037_15413389_1 . . . 84
3.3654 so_15413037_15413389_2 . . . 84
3.3655 so_15413037_15413389_0 . . . 84
3.3656 so_15413037_15413389_3 . . . 84
3.3657 so_9929703_9929734_3 . . . 84
3.3658 so_12374746_12375019_0 . . . 84
3.3659 so_15501470_15504781_2 . . . 84
3.3660 so_30352301_30352638_0 . . . 84
3.3661 so_11543625_11544332_0 . . . 84
3.3662 so_29467795_29467822_0 . . . 84
3.3663 so_7595541_7595654_1 . . . 84
3.3664 so_35337869_35337962_0 . . . 84
3.3665 so_35765449_35765500_0 . . . 84
3.3666 so_17432891_17433001_0 . . . 84
3.3667 so_34488559_34488782_3 . . . 84
3.3668 so_34488559_34488782_2 . . . 84
3.3669 so_29467795_29467822_1 . . . 84
3.3670 so_29467795_29467862_1 . . . 84
3.3671 so_27770228_27770450_1 . . . 84
3.3672 so_27770228_27770450_0 . . . 84
3.3673 Lvalues and rvalues . . . 85
3.3674 so_2220230_43197343_1 . . . 85
3.3675 so_8598498_8598574_0 . . . 85
3.3676 so_3601602_3601748_6 . . . 85
3.3677 so_3601602_34901787_2 . . . 85
3.3678 so_3601602_34901787_1 . . . 85
3.3679 so_3601602_3601748_5 . . . 85
3.3680 so_3601602_3601748_1 . . . 85
3.3681 so_3601602_3601748_3 . . . 85
3.3682 so_3601602_3601748_4 . . . 85
3.3683 so_3601602_3601748_2 . . . 85
3.3684 so_29434001_29434109_1 . . . 85
3.3685 so_9747010_9747062_1 . . . 85
3.3686 so_4285895_20614158_5 . . . 85
3.3687 so_31353610_31354103_0 . . . 85
3.3688 so_6865263_6865301_0 . . . 85
3.3689 so_42043414_42043414_3 . . . 85
3.3690 so_42043414_42043414_0 . . . 85
3.3691 so_20467514_20469655_2 . . . 85

3.3692 so_3007728_11237362_0 . . . 85
3.3693 so_3007728_11237362_1 . . . 85
3.3694 so_3007728_3007728_0 . . . 85
3.3695 so_7291874_7291895_0 . . . 85
3.3696 so_43521206_43550283_0 . . . 85
3.3697 so_26085820_26085890_0 . . . 85
3.3698 so_26085820_26085890_2 . . . 85
3.3699 so_26085820_26085890_1 . . . 85
3.3700 so_9152798_9152814_0 . . . 85
3.3701 so_14935722_14991772_0 . . . 85
3.3702 so_26321248_26321248_1 . . . 85
3.3703 so_30048135_30050906_0 . . . 85
3.3704 so_25674834_25674834_0 . . . 85
3.3705 so_25674834_25676435_0 . . . 85
3.3706 so_25674834_25674834_1 . . . 85
3.3707 so_43608823_43608969_0 . . . 85
3.3708 so_20443262_20443362_5 . . . 85
3.3709 so_3213591_3213598_0 . . . 85
3.3710 so_26290598_26290598_0 . . . 85
3.3711 so_34870498_34870498_0 . . . 85
3.3712 so_24672097_24673071_1 . . . 85
3.3713 so_31362252_31362290_2 . . . 85
3.3714 so_31362252_31362290_0 . . . 85
3.3715 so_17508703_17637679_1 . . . 85
3.3716 so_17508703_17637679_2 . . . 85
3.3717 so_6376580_6376764_4 . . . 85
3.3718 so_6376580_6376764_3 . . . 85
3.3719 so_31520406_31520617_0 . . . 85
3.3720 so_8804612_8808865_1 . . . 85
3.3721 so_19620835_19621107_0 . . . 85
3.3722 so_3612554_3612696_0 . . . 85
3.3723 so_27364787_27364969_0 . . . 85
3.3724 so_27364787_27364787_0 . . . 85
3.3725 so_27364787_27364899_0 . . . 85
3.3726 so_27364787_27364899_1 . . . 85
3.3727 so_11581903_11581903_0 . . . 85
3.3728 so_38926753_38926870_1 . . . 85
3.3729 so_38926753_38926870_0 . . . 85
3.3730 so_24518517_24519164_0 . . . 85
3.3731 so_25060794_25060988_0 . . . 85
3.3732 so_41331640_41353091_5 . . . 85
3.3733 so_25987133_25987392_1 . . . 85
3.3734 so_32238645_32239072_1 . . . 85
3.3735 so_15036281_15036281_0 . . . 85
3.3736 so_34315772_34315889_0 . . . 85
3.3737 so_20850536_20999389_3 . . . 85
3.3738 so_14991219_14991297_0 . . . 85
3.3739 so_14991219_14991219_0 . . . 85
3.3740 so_29116751_29116887_3 . . . 85
3.3741 so_29116751_29116887_1 . . . 85
3.3742 so_29116751_29116887_2 . . . 85

3.3743 so_6085531_19257003_2 . . . 85
3.3744 so_14189709_14189818_2 . . . 85
3.3745 so_6634840_6635848_0 . . . 85
3.3746 so_5924870_5924870_1 . . . 85
3.3747 so_14935722_17514157_2 . . . 85
3.3748 so_7460410_7460424_0 . . . 85
3.3749 so_5099769_5099769_0 . . . 85
3.3750 so_6376580_6376764_6 . . . 85
3.3751 so_6376580_6376699_0 . . . 85
3.3752 so_6376580_6376875_1 . . . 85
3.3753 so_20850536_20850536_0 . . . 85
3.3754 so_6376580_6376764_7 . . . 86
3.3755 so_18502188_18502590_0 . . . 86
3.3756 so_8585924_8587027_2 . . . 86
3.3757 so_46162766_46163082_1 . . . 86
3.3758 so_12160782_12160964_0 . . . 86
3.3759 so_39894005_39894089_0 . . . 86
3.3760 so_15824750_15825240_0 . . . 86
3.3761 so_26085820_26085890_3 . . . 86
3.3762 so_6376580_6376764_9 . . . 86
3.3763 so_35378090_35379796_0 . . . 86
3.3764 so_3232582_3233045_3 . . . 86
3.3765 so_7807437_7807667_0 . . . 86
3.3766 so_7807437_7807490_1 . . . 86
3.3767 so_7807437_7807490_2 . . . 86
3.3768 so_30521292_30521417_1 . . . 86
3.3769 so_6376580_6376764_10 . . . 86
3.3770 so_6376580_6376764_11 . . . 86
3.3771 so_6376580_6376764_12 . . . 86
3.3772 so_6376580_6376764_13 . . . 86
3.3773 so_32279647_32281264_4 . . . 86
3.3774 so_8546243_8546339_0 . . . 86
3.3775 so_30584706_30585119_0 . . . 86
3.3776 so_13782742_13782759_2 . . . 86
3.3777 so_44338559_44338825_4 . . . 86
3.3778 so_44338559_44338825_5 . . . 86
3.3779 so_44338559_44338825_2 . . . 86
3.3780 so_44338559_44338825_0 . . . 86
3.3781 so_44338559_44338825_1 . . . 86
3.3782 so_44338559_44338825_7 . . . 86
3.3783 so_44338559_44338825_6 . . . 86
3.3784 so_41096733_41096869_0 . . . 86
3.3785 so_40070352_40070428_0 . . . 86
3.3786 so_29066335_29066655_3 . . . 86
3.3787 so_29066335_29066655_1 . . . 86
3.3788 so_27610922_27610946_0 . . . 86
3.3789 so_15283523_15284276_9 . . . 86
3.3790 so_18475338_18475470_2 . . . 86
3.3791 so_7291874_7291895_1 . . . 86
3.3792 so_25308761_25308909_4 . . . 86
3.3793 so_25308761_25308881_10 . . . 86

3.3794 so_25308761_25308909_7 . . . 86
3.3795 so_25308761_25308909_5 . . . 86
3.3796 so_25308761_25308881_11 . . . 86
3.3797 so_25308761_25308909_8 . . . 86
3.3798 so_25308761_25308909_6 . . . 86
3.3799 so_25308761_25308881_4 . . . 86
3.3800 so_25308761_25308881_9 . . . 86
3.3801 so_25308761_25308909_1 . . . 86
3.3802 so_25308761_25308881_8 . . . 86
3.3803 so_25308761_25308909_3 . . . 86
3.3804 so_25308761_25308881_6 . . . 86
3.3805 so_25308761_25308881_7 . . . 86
3.3806 so_13170189_13170246_0 . . . 86
3.3807 so_25091602_25092726_0 . . . 86
3.3808 so_30048135_30050906_1 . . . 86
3.3809 so_36225493_36225493_0 . . . 86
3.3810 so_23140259_23140953_0 . . . 86
3.3811 so_15078638_15172304_0 . . . 86
3.3812 so_43608823_43608969_5 . . . 86
3.3813 so_43608823_43608969_4 . . . 86
3.3814 so_43608823_43608969_3 . . . 86
3.3815 so_14730896_14732625_3 . . . 86
3.3816 so_14730896_14732625_0 . . . 86
3.3817 so_14730896_14732625_4 . . . 86
3.3818 so_14730896_14732625_5 . . . 86
3.3819 so_24672097_24673071_0 . . . 86
3.3820 so_45041045_45052333_12 . . . 86
3.3821 so_45041045_45052333_9 . . . 86
3.3822 so_45041045_45052333_7 . . . 86
3.3823 so_45041045_45052333_14 . . . 86
3.3824 so_45041045_45052333_11 . . . 86
3.3825 so_45041045_45052333_13 . . . 86
3.3826 so_38744365_38744467_1 . . . 86
3.3827 so_38744365_38744467_3 . . . 86
3.3828 so_38744365_38744467_6 . . . 86
3.3829 so_38744365_38744467_2 . . . 86
3.3830 so_38744365_38744467_4 . . . 86
3.3831 so_38744365_38744467_5 . . . 86
3.3832 so_38744365_38744467_0 . . . 86
3.3833 so_46323266_46323266_1 . . . 86
3.3834 so_42134181_42134629_0 . . . 86
3.3835 so_39872246_39873599_3 . . . 86
3.3836 so_6376580_6376764_14 . . . 86
3.3837 so_47498585_47499092_0 . . . 86
3.3838 so_33069102_33277347_3 . . . 86
3.3839 so_37118426_37119041_3 . . . 86
3.3840 so_37118426_37119041_2 . . . 86
3.3841 so_37118426_37119041_0 . . . 86
3.3842 so_37118426_37119041_6 . . . 86
3.3843 so_37118426_37119041_4 . . . 86
3.3844 so_37118426_37119041_5 . . . 86

3.3845 so_37118426_37119041_7 . . . 86
3.3846 so_33849613_33849737_0 . . . 86
3.3847 so_6514663_6514767_0 . . . 86
3.3848 so_6514663_6514750_3 . . . 86
3.3849 so_6514663_6514750_5 . . . 86
3.3850 so_6514663_6514750_4 . . . 86
3.3851 so_6514663_6514767_1 . . . 86
3.3852 so_20275322_20275322_0 . . . 86
3.3853 so_20275322_20275322_3 . . . 86
3.3854 so_12612488_12615861_0 . . . 86
3.3855 so_12612488_12614819_0 . . . 86
3.3856 so_12612488_12613099_1 . . . 86
3.3857 so_4807643_4807726_0 . . . 86
3.3858 so_4807643_4807726_6 . . . 86
3.3859 so_4807643_4807726_4 . . . 86
3.3860 so_4807643_4807726_2 . . . 86
3.3861 so_4807643_4807726_3 . . . 86
3.3862 so_4807643_4807726_5 . . . 86
3.3863 so_34879858_34912703_6 . . . 86
3.3864 so_34879858_34912703_5 . . . 86
3.3865 so_34879858_34912703_8 . . . 86
3.3866 so_34879858_34912703_7 . . . 86
3.3867 so_34879858_34912703_1 . . . 86
3.3868 so_39603113_39603397_0 . . . 86
3.3869 so_18624449_18629436_0 . . . 86
3.3870 so_45136738_45137750_0 . . . 86
3.3871 so_42178179_42190913_2 . . . 86
3.3872 so_29121176_29122056_0 . . . 86
3.3873 so_29121176_29241728_0 . . . 86
3.3874 so_29121176_29217925_8 . . . 86
3.3875 so_29121176_29217925_11 . . . 86
3.3876 so_21176193_21176193_3 . . . 86
3.3877 so_15277819_15278030_0 . . . 86
3.3878 so_27492435_27492435_0 . . . 86
3.3879 so_39904101_39904258_2 . . . 86
3.3880 so_10965325_10965371_4 . . . 86
3.3881 so_33845965_33845965_0 . . . 86
3.3882 so_37418412_37418655_0 . . . 86
3.3883 so_39472903_39473175_1 . . . 86
3.3884 so_29317981_29318354_0 . . . 86
3.3885 so_36457841_36457841_0 . . . 86
3.3886 so_23829696_23831795_0 . . . 86
3.3887 so_44515378_44515398_4 . . . 86
3.3888 so_44515378_44515398_5 . . . 86
3.3889 so_44515378_44515398_3 . . . 86
3.3890 so_1863069_1863093_4 . . . 86
3.3891 so_36021322_36021322_0 . . . 86
3.3892 so_37481434_37481628_5 . . . 86
3.3893 so_37481434_37481628_3 . . . 86
3.3894 so_37481434_37481628_7 . . . 86
3.3895 so_37481434_37481628_0 . . . 86

3.3896 so_37481434_37481628_2 . . . 86
3.3897 so_37481434_37481628_6 . . . 86
3.3898 so_37481434_37481628_4 . . . 86
3.3899 so_37323232_37368823_0 . . . 86
3.3900 so_16260033_16260033_0 . . . 86
3.3901 so_31436620_31436956_7 . . . 86
3.3902 so_31436620_31436956_3 . . . 86
3.3903 so_31436620_31436956_6 . . . 86
3.3904 so_31436620_31436956_4 . . . 86
3.3905 so_31436620_31436956_5 . . . 86
3.3906 so_31436620_31436956_1 . . . 86
3.3907 so_14719529_16641131_0 . . . 86
3.3908 so_19757653_19757653_0 . . . 86
3.3909 so_43551665_43551805_1 . . . 86
3.3910 so_43152787_43153258_1 . . . 86
3.3911 so_43152787_43153258_2 . . . 86
3.3912 so_41487696_41488396_2 . . . 86
3.3913 so_9585990_9588075_0 . . . 86
3.3914 so_9585990_9588075_1 . . . 86
3.3915 so_14952504_14954413_1 . . . 86
3.3916 so_46044682_46044783_2 . . . 86
3.3917 so_46044682_46044783_1 . . . 86
3.3918 so_98650_7005988_3 . . . 86
3.3919 so_98650_7005988_0 . . . 86
3.3920 so_98650_7005988_1 . . . 86
3.3921 so_18659427_18659427_0 . . . 86
3.3922 so_9964418_9964418_0 . . . 86
3.3923 so_7321513_7321802_1 . . . 86
3.3924 Alignment . . . 87
3.3925 so_22586752_22587296_0 . . . 87
3.3926 so_32700032_32700393_0 . . . 87
3.3927 so_32700032_32700032_3 . . . 87
3.3928 so_5397447_5397599_0 . . . 87
3.3929 so_41385355_41388497_0 . . . 87
3.3930 so_32599788_32601159_1 . . . 87
3.3931 so_24831435_24831570_0 . . . 87
3.3932 so_13077086_13077395_0 . . . 87
3.3933 so_7191650_7191650_0 . . . 87
3.3934 so_7501211_7501423_0 . . . 87
3.3935 so_24788262_24788755_0 . . . 87
3.3936 so_7552305_7552305_0 . . . 87
3.3937 so_36805521_36805521_2 . . . 87
3.3938 so_9831445_9831732_0 . . . 87
3.3939 so_46425250_46426307_0 . . . 87
3.3940 so_4637774_4639568_1 . . . 87
3.3941 so_25058081_25058081_1 . . . 87
3.3942 so_26200639_26201073_2 . . . 87
3.3943 so_15833683_15833683_0 . . . 87
3.3944 so_24683142_24684529_2 . . . 87
3.3945 so_20876606_20876606_0 . . . 87
3.3946 so_32589328_32590117_0 . . . 87

3.3947 so_29340160_29342850_2 . . . 87
3.3948 so_38650310_38701636_0 . . . 87
3.3949 so_17091382_17094266_0 . . . 87
3.3950 so_43216468_43217371_0 . . . 87
3.3951 so_15523537_15527364_0 . . . 87
3.3952 so_15283523_15284276_3 . . . 87
3.3953 so_32700032_32700414_0 . . . 87
3.3954 so_32700032_32700393_2 . . . 87
3.3955 so_32700032_32700393_1 . . . 87
3.3956 so_32700032_32700414_1 . . . 87
3.3957 so_4827674_4827900_0 . . . 87
3.3958 so_35081546_35082217_5 . . . 87
3.3959 so_24045850_24046172_0 . . . 87
3.3960 so_31436620_31483849_2 . . . 87
3.3961 so_20876606_20876606_1 . . . 87
3.3962 so_8456236_8458776_3 . . . 87
3.3963 so_15523537_15527364_1 . . . 87
3.3964 so_10587879_10588226_1 . . . 87
3.3965 so_15459316_15481556_0 . . . 87
3.3966 so_27217656_27217701_1 . . . 87
3.3967 so_34376921_34377119_4 . . . 87
3.3968 so_24045850_24046172_1 . . . 87
3.3969 so_13097268_13100870_3 . . . 87
3.3970 so_38650310_38701636_3 . . . 87
3.3971 so_46398162_46398322_1 . . . 87
3.3972 so_34376921_34377119_3 . . . 87
3.3973 so_35081546_35082217_6 . . . 87
3.3974 so_8456236_8458776_0 . . . 87
3.3975 so_42523833_42524024_0 . . . 87
3.3976 so_3255671_3256565_1 . . . 87
3.3977 so_7054176_7054354_3 . . . 87
3.3978 so_15833683_15833683_1 . . . 87
3.3979 so_38650310_38701636_2 . . . 87
3.3980 so_8456236_8458776_1 . . . 87
3.3981 so_42523833_42524024_2 . . . 87
3.3982 so_36805521_36805521_1 . . . 87
3.3983 so_13097268_13097268_1 . . . 87
3.3984 so_15833683_15833683_3 . . . 87
3.3985 so_15833683_15833683_5 . . . 87
3.3986 so_15833683_15833683_4 . . . 87
3.3987 so_15833683_15833683_2 . . . 87
3.3988 so_19251360_19252773_0 . . . 88
3.3989 so_26200639_26201073_3 . . . 88
3.3990 so_13097268_13100870_4 . . . 88
3.3991 so_5216071_5216238_1 . . . 88

4 Standard conversions **89**
4.1 so_34486886_34486985_2 . . . 89
4.2 so_34486886_34486985_4 . . . 89
4.3 so_34486886_34486985_1 . . . 89
4.4 so_34486886_34486985_3 . . . 89

4.5 so_5854581_5854862_2 . . . 89
4.6 so_5854581_5854862_3 . . . 89
4.7 so_37494170_37494202_2 . . . 89
4.8 so_37494170_37494202_1 . . . 89
4.9 so_29610318_29611071_1 . . . 89
4.10 so_13092266_13092320_2 . . . 89
4.11 so_13092266_13092320_3 . . . 89
4.12 so_13092266_13092320_0 . . . 89
4.13 so_13092266_13092320_1 . . . 89
4.14 so_32522701_32523222_0 . . . 89
4.15 so_16726014_16738704_2 . . . 89
4.16 so_16726014_16738704_6 . . . 89
4.17 so_12396132_12396208_6 . . . 89
4.18 so_26692637_26692775_3 . . . 89
4.19 so_35790664_35790867_3 . . . 89
4.20 so_35790664_35790867_1 . . . 89
4.21 so_35790664_35790867_2 . . . 89
4.22 so_35790664_35790867_0 . . . 89
4.23 so_6401818_6402127_1 . . . 89
4.24 so_6401818_6402127_4 . . . 89
4.25 so_6401818_6402127_2 . . . 89
4.26 so_6401818_6402127_3 . . . 89
4.27 so_21053273_21053273_1 . . . 89
4.28 so_38772637_38772848_1 . . . 89
4.29 so_38772637_38772848_2 . . . 89
4.30 so_32141754_32141860_3 . . . 89
4.31 so_28156381_28159822_0 . . . 89
4.32 so_23530206_23530445_0 . . . 89
4.33 so_17501942_17503313_5 . . . 89
4.34 so_17501942_17503313_6 . . . 89
4.35 so_28862203_28862234_5 . . . 89
4.36 so_28862203_28862234_4 . . . 89
4.37 so_28862203_28862503_3 . . . 89
4.38 so_28862203_28862503_0 . . . 89
4.39 so_28862203_28862234_2 . . . 89
4.40 so_28862203_28862234_1 . . . 89
4.41 so_28862203_28862503_2 . . . 89
4.42 so_28862203_28862503_1 . . . 89
4.43 so_28862203_28862234_6 . . . 89
4.44 so_28862203_28862234_3 . . . 89
4.45 so_18405501_18405796_1 . . . 89
4.46 so_12279601_12279842_3 . . . 89
4.47 so_4316316_4322041_0 . . . 89
4.48 so_4316316_4322041_2 . . . 89
4.49 so_32253788_32254282_0 . . . 89
4.50 so_17384193_17384226_3 . . . 89
4.51 so_39401136_39401437_0 . . . 89
4.52 so_10775361_10775667_4 . . . 89
4.53 so_28847468_28847809_2 . . . 89
4.54 so_28847468_28847809_0 . . . 89
4.55 so_25540033_25540631_1 . . . 89

4.56 so_25540033_25540631_0 . . . 89
4.57 so_26954276_26955983_0 . . . 89
4.58 so_26954276_26955983_4 . . . 89
4.59 so_26954276_26955983_1 . . . 89
4.60 so_26954276_26955983_2 . . . 89
4.61 so_26954276_26955983_3 . . . 89
4.62 so_26954276_26955983_5 . . . 89
4.63 so_30490592_30490916_1 . . . 89
4.64 so_12419384_12419609_1 . . . 89
4.65 so_12419384_12419609_0 . . . 89
4.66 so_24079001_24079001_2 . . . 89
4.67 so_22243721_22243785_1 . . . 89
4.68 so_22243721_22243785_2 . . . 89
4.69 so_34147715_34147799_0 . . . 89
4.70 so_45322138_45322372_0 . . . 89
4.71 so_45322138_45322372_1 . . . 89
4.72 so_13366290_13366758_2 . . . 89
4.73 so_17665110_17665110_1 . . . 89
4.74 so_12372442_12373830_2 . . . 89
4.75 so_37494170_37494202_3 . . . 89
4.76 so_25395605_25395605_0 . . . 89
4.77 so_1445274_1445327_2 . . . 89
4.78 so_27054485_27054786_1 . . . 89
4.79 so_23003942_23013978_2 . . . 89
4.80 so_18405501_18405796_2 . . . 89
4.81 so_21405674_21407037_0 . . . 89
4.82 so_13434672_13543853_4 . . . 89
4.83 so_17220498_17220579_0 . . . 89
4.84 so_25241516_25242022_4 . . . 89
4.85 so_28074071_28074071_1 . . . 89
4.86 so_46296448_46297037_2 . . . 89
4.87 so_8013599_8014548_1 . . . 89
4.88 so_44679200_44679200_0 . . . 89
4.89 so_14720913_14721483_2 . . . 89
4.90 so_23638282_23638529_1 . . . 89
4.91 so_36392600_36392775_0 . . . 89
4.92 so_38535292_38535667_1 . . . 89
4.93 so_35626950_35627300_0 . . . 89
4.94 so_15876960_15878243_4 . . . 89
4.95 so_6242768_6242768_0 . . . 89
4.96 so_25047109_25048219_2 . . . 89
4.97 so_42613866_42614016_0 . . . 89
4.98 so_42613866_42613928_1 . . . 89
4.99 so_22243721_22243785_5 . . . 89
4.100 so_39563948_39564002_0 . . . 89
4.101 so_25143860_25144037_2 . . . 89
4.102 so_25143860_25143950_2 . . . 89
4.103 so_34371901_34372209_1 . . . 89
4.104 so_34360286_34577436_1 . . . 89
4.105 so_25047109_25048348_1 . . . 89
4.106 so_25047109_25048219_1 . . . 89

4.107 so_25047109_25048219_10 . . . 89
4.108 so_42438179_42438179_0 . . . 90
4.109 so_37494170_37494202_4 . . . 90
4.110 so_28346426_28346853_1 . . . 90
4.111 so_14935722_17514157_1 . . . 90
4.112 so_14935722_17514157_0 . . . 90
4.113 so_27054485_27054786_2 . . . 90
4.114 so_33433337_33433519_0 . . . 90
4.115 so_12419384_12419609_2 . . . 90
4.116 so_38545714_38545714_0 . . . 90
4.117 so_17453738_28557688_4 . . . 90
4.118 so_13385601_13385601_0 . . . 90
4.119 so_40909095_40911798_0 . . . 90
4.120 so_24867294_24868125_0 . . . 90
4.121 so_37043401_37043401_0 . . . 90
4.122 so_34490247_34490247_0 . . . 90
4.123 so_34490247_34490537_2 . . . 90
4.124 so_34834986_34838255_1 . . . 90
4.125 so_4316316_4322041_4 . . . 90
4.126 so_14935722_14991772_2 . . . 90
4.127 Lvalue-to-rvalue conversion . . . 90
4.128 so_8016780_28846608_2 . . . 90
4.129 so_13981993_13982675_0 . . . 90
4.130 so_25674834_25693147_0 . . . 90
4.131 so_15030379_15030445_5 . . . 90
4.132 so_11331308_11331662_0 . . . 90
4.133 so_27007322_27007359_1 . . . 90
4.134 so_46065337_46065337_1 . . . 90
4.135 so_46509077_46509077_0 . . . 90
4.136 so_36457841_36457841_4 . . . 90
4.137 so_29868792_29873914_5 . . . 90
4.138 so_20850536_20999389_4 . . . 90
4.139 so_14024639_14024679_0 . . . 90
4.140 so_17636464_17636464_0 . . . 90
4.141 so_14659752_14659931_11 . . . 90
4.142 so_39211727_39212207_1 . . . 90
4.143 so_42128911_42129251_5 . . . 90
4.144 so_42128911_42129251_1 . . . 90
4.145 so_42128911_42129251_2 . . . 90
4.146 so_42128911_42129251_6 . . . 90
4.147 so_14935722_14991772_11 . . . 90
4.148 so_14935722_14991772_8 . . . 90
4.149 so_14935722_14991772_6 . . . 90
4.150 so_14935722_14991772_9 . . . 90
4.151 so_14935722_14991772_12 . . . 90
4.152 so_14935722_14991772_7 . . . 90
4.153 so_28727439_37382133_0 . . . 90
4.154 so_7460410_7460424_1 . . . 90
4.155 so_8943261_8943287_7 . . . 90
4.156 so_33395724_33395724_2 . . . 90
4.157 so_26290598_26290598_2 . . . 90

4.158 so_6376580_6376743_0 . . . 90
4.159 so_6376580_6376699_1 . . . 90
4.160 so_30694069_30694084_3 . . . 90
4.161 so_46065337_46065337_2 . . . 90
4.162 so_25999309_25999455_1 . . . 90
4.163 so_46509077_46509077_2 . . . 90
4.164 so_46509077_46509077_5 . . . 90
4.165 so_46509077_46509077_4 . . . 90
4.166 so_46509077_46509077_1 . . . 90
4.167 so_46509077_46509077_3 . . . 90
4.168 so_988158_47169070_0 . . . 90
4.169 so_36457841_36457841_6 . . . 90
4.170 so_36457841_36457841_5 . . . 90
4.171 so_28506342_28506460_1 . . . 90
4.172 so_28506342_28506460_2 . . . 90
4.173 so_28506342_28506460_3 . . . 90
4.174 so_29868792_29873914_6 . . . 90
4.175 so_20850536_20999389_5 . . . 90
4.176 so_20850536_20999389_6 . . . 90
4.177 so_23763748_24067465_2 . . . 90
4.178 so_37043401_37055970_4 . . . 90
4.179 so_37043401_37055970_3 . . . 90
4.180 Array-to-pointer conversion . . . 91
4.181 so_29454806_29454806_1 . . . 91
4.182 so_33202400_33202446_0 . . . 91
4.183 so_15458883_15458883_0 . . . 91
4.184 so_41461175_41461201_2 . . . 91
4.185 so_32995010_32995212_1 . . . 91
4.186 so_42761285_42761747_0 . . . 91
4.187 so_20467514_20469655_1 . . . 91
4.188 so_1461432_1462103_1 . . . 91
4.189 so_27375117_27375117_1 . . . 91
4.190 so_28680325_28680414_0 . . . 91
4.191 so_24478830_24480354_2 . . . 91
4.192 so_18145567_18146631_1 . . . 91
4.193 so_33647319_33647779_0 . . . 91
4.194 so_13808932_13808933_11 . . . 91
4.195 so_27450021_27451656_1 . . . 91
4.196 so_36435302_36435400_1 . . . 91
4.197 so_1370399_1370451_0 . . . 91
4.198 so_25988412_25988540_0 . . . 91
4.199 so_28780307_28780498_1 . . . 91
4.200 so_28780479_28780720_0 . . . 91
4.201 so_24662385_24662879_1 . . . 91
4.202 so_3655184_3655381_0 . . . 91
4.203 so_36051008_36101569_1 . . . 91
4.204 so_15032044_15032114_5 . . . 91
4.205 so_45056234_45056630_1 . . . 91
4.206 so_32253788_32254282_1 . . . 91
4.207 so_4223617_4223617_0 . . . 91
4.208 so_6443230_6444871_2 . . . 91

4.209 so_28119793_28119895_0 . . . 91
4.210 so_11198776_11198871_0 . . . 91
4.211 so_23428684_23428685_3 . . . 91
4.212 so_12625679_12625732_1 . . . 91
4.213 so_33729176_33733204_0 . . . 91
4.214 so_26441950_26442287_0 . . . 91
4.215 so_26724674_26725089_4 . . . 91
4.216 so_41212904_41256180_0 . . . 91
4.217 so_3959705_3959761_0 . . . 91
4.218 so_23967553_23967656_0 . . . 91
4.219 so_15943514_15943544_2 . . . 91
4.220 Function-to-pointer conversion . . . 91
4.221 so_27594273_27594665_2 . . . 91
4.222 so_27594273_27594665_1 . . . 91
4.223 so_28086502_28087648_1 . . . 91
4.224 so_28086502_28087648_2 . . . 91
4.225 so_24434157_24630425_1 . . . 91
4.226 so_26559758_30424903_0 . . . 91
4.227 so_41097868_41098916_1 . . . 91
4.228 so_37501982_37502106_1 . . . 91
4.229 so_37501982_37502106_2 . . . 91
4.230 so_2795575_2795803_0 . . . 91
4.231 so_23530206_23530445_1 . . . 91
4.232 so_18283612_18283612_1 . . . 91
4.233 so_26173403_26173403_2 . . . 91
4.234 so_31643245_31643450_1 . . . 91
4.235 so_24386558_24387253_1 . . . 91
4.236 so_34433172_34433523_0 . . . 91
4.237 so_34433172_34433523_1 . . . 91
4.238 so_35654908_35655727_1 . . . 91
4.239 so_16626033_16626055_0 . . . 91
4.240 so_27226680_27226680_0 . . . 91
4.241 so_28660352_28660610_1 . . . 91
4.242 so_7245829_7245858_0 . . . 91
4.243 so_27869835_27869972_1 . . . 91
4.244 so_27869835_27869972_0 . . . 91
4.245 so_7761527_7761536_0 . . . 91
4.246 Qualification conversions . . . 91
4.247 so_28680325_28680414_2 . . . 91
4.248 so_24555288_24555400_0 . . . 91
4.249 so_17471497_17471602_6 . . . 91
4.250 so_7595541_7595541_0 . . . 91
4.251 so_5654345_5654540_0 . . . 91
4.252 so_5654345_5654432_0 . . . 91
4.253 so_12660845_12660897_1 . . . 91
4.254 so_60000_2688631_3 . . . 91
4.255 so_19311239_19311738_2 . . . 91
4.256 so_40200807_40200807_0 . . . 91
4.257 so_21213881_21214761_0 . . . 91
4.258 so_23371626_23372137_0 . . . 91
4.259 so_17460149_17460149_2 . . . 91

4.260 so_17460149_17460149_1 . . . 91
4.261 so_17460149_17460149_0 . . . 91
4.262 Integral promotions . . . 92
4.263 so_18195312_18195408_6 . . . 92
4.264 so_41720176_41720882_1 . . . 92
4.265 so_24489929_24489971_3 . . . 92
4.266 so_32530475_32530759_1 . . . 92
4.267 so_33731158_33731465_1 . . . 92
4.268 so_7538448_7538579_0 . . . 92
4.269 so_37055444_37056269_2 . . . 92
4.270 so_37055444_37056269_1 . . . 92
4.271 so_30361112_30363234_1 . . . 92
4.272 so_4814668_4814901_3 . . . 92
4.273 so_40785566_40785774_1 . . . 92
4.274 so_39856970_39856970_0 . . . 92
4.275 so_39856970_39863945_5 . . . 92
4.276 so_3357614_3357684_5 . . . 92
4.277 so_34710492_34710865_1 . . . 92
4.278 so_11985774_12338977_21 . . . 92
4.279 so_11985774_12338977_3 . . . 92
4.280 so_27844971_27877127_3 . . . 92
4.281 so_38772637_38773075_0 . . . 92
4.282 so_38772637_38773012_0 . . . 92
4.283 so_32684369_32684432_3 . . . 92
4.284 so_3100365_3100936_1 . . . 92
4.285 so_30473958_30474166_2 . . . 92
4.286 so_14695118_14695400_6 . . . 92
4.287 so_24079001_24079229_2 . . . 92
4.288 so_24079001_24079229_1 . . . 92
4.289 so_27931157_27931379_2 . . . 92
4.290 so_20031343_20031372_0 . . . 92
4.291 so_8425118_8425141_2 . . . 92
4.292 so_4267194_4267258_1 . . . 92
4.293 so_12889237_12889697_2 . . . 92
4.294 so_21242070_21242070_1 . . . 92
4.295 so_34252403_34252511_2 . . . 92
4.296 so_45775764_45776027_1 . . . 92
4.297 so_14745825_14746323_2 . . . 92
4.298 so_20192064_20192160_0 . . . 92
4.299 so_1280304_1280362_2 . . . 92
4.300 so_26101257_26101357_1 . . . 92
4.301 so_7131764_7131815_0 . . . 92
4.302 so_19471235_19471347_0 . . . 92
4.303 so_32271852_32273373_1 . . . 92
4.304 so_28235805_28235963_2 . . . 92
4.305 so_38582749_38582749_0 . . . 92
4.306 so_24314638_24314853_8 . . . 92
4.307 so_24314638_24314853_7 . . . 92
4.308 so_20389822_20390174_0 . . . 92
4.309 so_47336098_47336098_1 . . . 92
4.310 so_30645436_30645436_1 . . . 92

4.311 so_8713490_8713579_4 . . . 92
4.312 so_46283690_46284103_1 . . . 92
4.313 so_5641805_5641928_1 . . . 92
4.314 so_29068333_29068918_1 . . . 92
4.315 so_28141871_28142480_6 . . . 92
4.316 so_4814668_4814901_4 . . . 92
4.317 so_31182316_31182470_0 . . . 92
4.318 so_6257807_6259317_2 . . . 92
4.319 so_34710492_34710865_2 . . . 92
4.320 so_11985774_12338977_4 . . . 92
4.321 so_38772637_38773012_3 . . . 92
4.322 so_28010565_28018488_0 . . . 92
4.323 so_20552491_20552672_0 . . . 92
4.324 so_14328034_14328034_0 . . . 92
4.325 so_42762610_42762659_0 . . . 92
4.326 so_25633040_25633075_1 . . . 92
4.327 so_25633040_25633075_2 . . . 92
4.328 so_1280304_1280362_3 . . . 92
4.329 so_14203654_14203654_1 . . . 92
4.330 so_15481532_15481828_2 . . . 92
4.331 so_15481532_15481828_1 . . . 92
4.332 so_22493879_22494119_10 . . . 92
4.333 so_4329277_4329360_1 . . . 92
4.334 so_17819109_17819235_2 . . . 92
4.335 so_22017190_22017585_0 . . . 92
4.336 so_30108274_30108422_2 . . . 92
4.337 so_34710492_34710865_3 . . . 92
4.338 so_23724244_23724337_1 . . . 92
4.339 so_13750296_13750503_2 . . . 92
4.340 so_32405505_32405618_1 . . . 92
4.341 so_14206403_14208457_0 . . . 92
4.342 so_17819109_17819235_0 . . . 92
4.343 so_17819109_17819109_0 . . . 92
4.344 so_18195312_18195408_5 . . . 93
4.345 so_19159379_19159481_0 . . . 93
4.346 so_37413895_37415174_0 . . . 93
4.347 so_34336024_34336157_0 . . . 93
4.348 so_34710492_34710865_4 . . . 93
4.349 so_35536146_35536263_0 . . . 93
4.350 so_35536146_35538979_0 . . . 93
4.351 so_35536146_35538979_1 . . . 93
4.352 so_21956017_21961992_1 . . . 93
4.353 so_37166131_37166556_9 . . . 93
4.354 so_44618373_44618474_0 . . . 93
4.355 so_14185352_14185499_1 . . . 93
4.356 so_14206403_14206403_2 . . . 93
4.357 so_4814668_4814901_5 . . . 93
4.358 so_34710492_34710865_5 . . . 93
4.359 so_11985774_12338977_22 . . . 93
4.360 so_1280304_1280362_4 . . . 93
4.361 so_12498709_12499124_0 . . . 93

4.362 so_33380742_33398698_9 . . . 93
4.363 so_21351450_21351606_0 . . . 93
4.364 so_21351450_21351616_2 . . . 93
4.365 so_14388855_14388887_1 . . . 93
4.366 so_4814668_4814901_6 . . . 93
4.367 so_9670854_9671195_0 . . . 93
4.368 so_28778706_28778747_0 . . . 93
4.369 so_30108274_30108422_3 . . . 93
4.370 so_24769567_24769567_1 . . . 93
4.371 so_37214420_37214756_0 . . . 93
4.372 so_33981801_33981801_0 . . . 93
4.373 so_32800866_32800889_1 . . . 93
4.374 so_18984934_18985085_3 . . . 93
4.375 so_22620517_22620776_1 . . . 93
4.376 so_29402367_29402657_1 . . . 93
4.377 so_4276207_4276232_2 . . . 93
4.378 so_4276207_4276239_0 . . . 93
4.379 so_1280304_1280362_5 . . . 93
4.380 so_21135697_21135954_0 . . . 93
4.381 so_26143976_26144079_5 . . . 93
4.382 so_26143976_26144237_3 . . . 93
4.383 so_47439616_47439700_1 . . . 93
4.384 so_23874077_23874224_1 . . . 93
4.385 so_30888851_30891329_3 . . . 93
4.386 so_38582749_38583542_1 . . . 93
4.387 so_33142145_33142227_0 . . . 93
4.388 so_33110423_33112433_0 . . . 93
4.389 Floating point promotion . . . 93
4.390 so_22019261_22019445_1 . . . 93
4.391 so_36840173_36840390_2 . . . 93
4.392 so_28097564_28097654_2 . . . 93
4.393 so_12396132_12396208_8 . . . 93
4.394 so_8757815_8758072_1 . . . 93
4.395 so_10775361_10775667_3 . . . 93
4.396 so_30490592_30490916_0 . . . 93
4.397 Integral conversions . . . 93
4.398 so_1461381_1462060_1 . . . 93
4.399 so_13569566_28453985_0 . . . 93
4.400 so_39683781_39683907_1 . . . 93
4.401 so_4640543_4640543_0 . . . 93
4.402 so_38772637_38773075_1 . . . 93
4.403 so_32684369_32684432_4 . . . 93
4.404 so_39872246_39873599_1 . . . 93
4.405 so_25909051_25909939_1 . . . 93
4.406 so_25490444_25491098_0 . . . 93
4.407 so_12786579_12786656_2 . . . 93
4.408 so_23070568_23070889_1 . . . 93
4.409 so_1751346_1751368_0 . . . 93
4.410 so_675976_676008_2 . . . 93
4.411 so_36991202_36991267_0 . . . 93
4.412 so_14345548_14346544_1 . . . 93

4.413 so_27209568_27209693_2 . . . 93
4.414 so_36439777_36440524_0 . . . 93
4.415 so_9467475_9468939_0 . . . 93
4.416 so_27833289_27833306_0 . . . 93
4.417 so_27109701_27110545_2 . . . 93
4.418 so_30048135_30048395_0 . . . 93
4.419 so_5458204_5458251_0 . . . 93
4.420 so_13122410_13122482_0 . . . 93
4.421 so_36583533_36583663_1 . . . 93
4.422 so_22801069_22801135_0 . . . 93
4.423 so_28182384_28182504_1 . . . 93
4.424 so_39872246_39873599_2 . . . 93
4.425 so_33307058_33307090_0 . . . 93
4.426 so_41276869_41276995_0 . . . 93
4.427 so_28196510_28197139_0 . . . 93
4.428 so_11419251_11419296_0 . . . 93
4.429 so_27931157_27931379_3 . . . 93
4.430 so_6752567_6752688_1 . . . 93
4.431 so_8888748_8888811_2 . . . 93
4.432 so_22131388_22131521_0 . . . 93
4.433 so_22131388_22131473_0 . . . 93
4.434 so_40396562_40396752_1 . . . 93
4.435 so_14065337_14065375_0 . . . 93
4.436 so_25490444_25491098_1 . . . 93
4.437 so_37798682_37800887_1 . . . 93
4.438 so_36182654_36183442_1 . . . 93
4.439 so_31677864_31677997_0 . . . 93
4.440 so_29784307_29784353_0 . . . 93
4.441 so_27405418_27405419_7 . . . 93
4.442 so_38857155_38862427_1 . . . 93
4.443 so_18506938_18507207_0 . . . 93
4.444 so_41058221_41058349_0 . . . 93
4.445 so_38737024_38738152_3 . . . 93
4.446 so_12786579_12786656_3 . . . 93
4.447 so_1229751_1551599_2 . . . 93
4.448 so_1229751_1551599_3 . . . 93
4.449 so_21483564_21483704_1 . . . 93
4.450 so_17313579_17313717_0 . . . 93
4.451 so_16826611_16826659_0 . . . 93
4.452 so_31203839_31203972_0 . . . 93
4.453 so_2711522_2711560_0 . . . 93
4.454 so_36293059_36293141_0 . . . 93
4.455 so_18994563_18994637_0 . . . 93
4.456 so_40721742_40722601_0 . . . 93
4.457 so_8208023_8208058_0 . . . 93
4.458 so_10048130_10048295_2 . . . 93
4.459 so_23070568_23070889_2 . . . 93
4.460 so_37583431_37583477_0 . . . 93
4.461 so_32477249_32479159_3 . . . 93
4.462 so_675976_676008_3 . . . 93
4.463 so_4243915_4243950_0 . . . 93

4.464 so_26323585_26323722_0 . . . 93
4.465 so_33436341_33436474_0 . . . 93
4.466 so_33436341_33436493_0 . . . 93
4.467 so_13569566_28453985_2 . . . 93
4.468 so_27209568_27209693_0 . . . 93
4.469 so_46264019_46264301_0 . . . 93
4.470 so_12125650_12125836_6 . . . 93
4.471 so_39683781_39683907_2 . . . 93
4.472 so_9467475_9468939_1 . . . 93
4.473 so_7601731_7602006_0 . . . 93
4.474 so_17398959_17437214_3 . . . 93
4.475 so_6257807_6259317_4 . . . 93
4.476 so_13122410_13122482_1 . . . 93
4.477 so_36583533_36583663_2 . . . 93
4.478 so_28779803_28779876_3 . . . 93
4.479 so_28182384_28182504_0 . . . 93
4.480 so_12467576_12467576_0 . . . 93
4.481 so_23008320_23010167_0 . . . 93
4.482 so_26284675_26285137_0 . . . 93
4.483 so_6752567_6752688_2 . . . 93
4.484 so_40396562_40396752_3 . . . 93
4.485 so_25490444_25491098_2 . . . 93
4.486 so_6770258_6770275_2 . . . 93
4.487 so_27405418_27405419_8 . . . 93
4.488 so_23844111_23844166_0 . . . 93
4.489 so_22623453_22624277_0 . . . 93
4.490 so_41058221_41058349_1 . . . 93
4.491 so_38737024_38738152_2 . . . 93
4.492 so_40707273_40707459_1 . . . 93
4.493 so_1229751_1551599_4 . . . 93
4.494 so_16826611_16826659_1 . . . 93
4.495 so_24826139_24826298_5 . . . 93
4.496 so_11372115_11372154_0 . . . 93
4.497 so_36293059_36293141_1 . . . 93
4.498 so_18994563_18994637_1 . . . 93
4.499 so_12343690_12343849_0 . . . 93
4.500 so_40721742_40722601_1 . . . 93
4.501 so_26331035_26331035_2 . . . 93
4.502 so_10048130_10048295_3 . . . 93
4.503 so_23070568_23070889_3 . . . 93
4.504 so_33110423_33112433_1 . . . 93
4.505 so_5369770_5369783_0 . . . 93
4.506 so_26521142_26521167_0 . . . 93
4.507 so_28778706_28778744_0 . . . 93
4.508 so_33981801_33982104_0 . . . 93
4.509 so_26438346_26438713_0 . . . 93
4.510 so_28447030_28447049_0 . . . 93
4.511 so_17418584_17418710_1 . . . 93
4.512 so_7661691_7661768_0 . . . 93
4.513 so_23070568_23070889_4 . . . 93
4.514 so_15481532_15481532_0 . . . 93

4.515 so_23070568_23070889_5 . . . 93
4.516 Floating point conversions . . . 93
4.517 so_22019261_22019445_3 . . . 93
4.518 so_36840173_36840390_3 . . . 93
4.519 so_12396132_12396208_9 . . . 93
4.520 so_31223666_31223711_3 . . . 93
4.521 so_31223666_31223711_2 . . . 93
4.522 so_25229832_25231230_1 . . . 93
4.523 so_37415723_37415877_0 . . . 93
4.524 so_8757815_8758072_2 . . . 93
4.525 so_10775361_10775667_6 . . . 93
4.526 so_17751031_17751118_0 . . . 93
4.527 so_28847468_28847809_3 . . . 93
4.528 so_26091955_26091993_1 . . . 93
4.529 so_12396132_12396208_10 . . . 94
4.530 so_8757815_8758072_3 . . . 94
4.531 so_10775361_10775667_7 . . . 94
4.532 Floating-integral conversions . . . 94
4.533 so_7703575_7703674_0 . . . 94
4.534 so_27540292_27540394_1 . . . 94
4.535 so_44253024_44253076_1 . . . 94
4.536 so_8251059_8251075_0 . . . 94
4.537 so_31748296_31749381_0 . . . 94
4.538 so_21165640_21165727_0 . . . 94
4.539 so_38443403_38443541_1 . . . 94
4.540 so_22957224_22957256_0 . . . 94
4.541 so_44443968_44444153_0 . . . 94
4.542 so_33824954_33825360_0 . . . 94
4.543 so_22652214_22652447_2 . . . 94
4.544 so_22652214_22652447_1 . . . 94
4.545 so_26081712_26081736_0 . . . 94
4.546 so_26081712_26081738_1 . . . 94
4.547 so_32684369_32684432_5 . . . 94
4.548 so_25005216_25028515_0 . . . 94
4.549 so_35225311_35225313_0 . . . 94
4.550 so_20421300_20421399_1 . . . 94
4.551 so_27007322_27007359_0 . . . 94
4.552 so_34597959_34597989_0 . . . 94
4.553 so_21478945_21488975_1 . . . 94
4.554 so_32810978_32810978_0 . . . 94
4.555 so_25113661_25113863_2 . . . 94
4.556 so_38795544_38796162_0 . . . 94
4.557 so_38795544_38795544_1 . . . 94
4.558 so_25857843_25857843_0 . . . 94
4.559 so_33829101_33829101_1 . . . 94
4.560 so_30851490_30851593_0 . . . 94
4.561 so_19079290_19079447_0 . . . 94
4.562 so_17225778_17225778_0 . . . 94
4.563 so_25691533_25717310_0 . . . 94
4.564 so_9158567_9159641_1 . . . 94
4.565 so_26091955_26091993_3 . . . 94

4.566 so_19345989_19346021_0 . . . 94
4.567 so_19345989_19346024_0 . . . 94
4.568 so_37818028_37826750_0 . . . 94
4.569 so_40694384_40694536_0 . . . 94
4.570 so_5670207_5671283_1 . . . 94
4.571 so_29773711_29773975_0 . . . 94
4.572 so_34856965_34857406_2 . . . 94
4.573 so_21574449_21574902_1 . . . 94
4.574 so_25541296_25541683_1 . . . 94
4.575 so_29932335_29932493_0 . . . 94
4.576 so_25857843_25857843_1 . . . 94
4.577 so_25691533_25717310_1 . . . 94
4.578 so_24423651_24423821_1 . . . 94
4.579 Pointer conversions . . . 94
4.580 so_2960496_2960528_0 . . . 94
4.581 so_39914051_39914347_0 . . . 94
4.582 so_25060360_25060360_0 . . . 94
4.583 so_1461381_1462060_3 . . . 94
4.584 so_6172232_6177338_0 . . . 94
4.585 so_4665709_4665867_0 . . . 94
4.586 so_32136092_32137530_3 . . . 94
4.587 so_32136092_32137530_1 . . . 94
4.588 so_37542316_37543100_1 . . . 94
4.589 so_37542316_37543100_0 . . . 94
4.590 so_37542316_37543100_2 . . . 94
4.591 so_34725939_34726797_1 . . . 94
4.592 so_34725939_34726797_3 . . . 94
4.593 so_15836499_15836511_4 . . . 94
4.594 so_31163449_31163665_0 . . . 94
4.595 so_24213159_24213223_0 . . . 94
4.596 so_22739071_22739113_0 . . . 94
4.597 so_19083709_19083981_3 . . . 94
4.598 so_9670854_42998291_0 . . . 94
4.599 so_8600486_8601976_0 . . . 94
4.600 so_28573215_28573215_4 . . . 94
4.601 so_37724861_37727181_1 . . . 94
4.602 so_41058178_41058695_0 . . . 94
4.603 so_41058178_41058695_2 . . . 94
4.604 so_23638282_23638529_3 . . . 94
4.605 so_37214420_37214756_1 . . . 94
4.606 so_37214420_37214756_2 . . . 94
4.607 so_2823598_2823624_0 . . . 94
4.608 so_25650592_25656153_4 . . . 94
4.609 so_33949124_33949180_0 . . . 94
4.610 so_9953068_9954287_2 . . . 94
4.611 so_32009694_32009989_1 . . . 94
4.612 so_17501942_17503313_3 . . . 94
4.613 so_35153996_35154862_1 . . . 94
4.614 so_25909051_25909289_0 . . . 94
4.615 so_33656742_33656869_2 . . . 94
4.616 so_16563114_16563381_2 . . . 94

4.617 so_35162259_35162401_0 . . . 94
4.618 so_7547147_7547183_0 . . . 94
4.619 so_40395066_40396015_1 . . . 94
4.620 so_4966002_4966034_3 . . . 94
4.621 so_2511921_2514087_0 . . . 94
4.622 so_21964120_21964142_1 . . . 94
4.623 so_8067897_8068024_0 . . . 94
4.624 so_5014061_5015089_1 . . . 94
4.625 so_28574069_28580509_0 . . . 94
4.626 so_28574069_28574093_0 . . . 94
4.627 so_41582864_41583216_0 . . . 94
4.628 so_27121395_27121591_2 . . . 94
4.629 so_27121395_27121512_1 . . . 94
4.630 so_27121395_27121591_1 . . . 94
4.631 so_16842660_16842753_0 . . . 94
4.632 so_12024706_12024724_5 . . . 94
4.633 so_15550203_15550366_0 . . . 94
4.634 so_21431868_21431949_0 . . . 94
4.635 so_40943080_40943130_1 . . . 94
4.636 so_27238670_27238859_0 . . . 94
4.637 so_24216202_24216284_0 . . . 94
4.638 so_24216202_24216202_0 . . . 94
4.639 so_19796413_19796549_0 . . . 94
4.640 so_22106713_22107408_0 . . . 94
4.641 so_28505090_28505277_1 . . . 94
4.642 so_11888581_11888673_0 . . . 94
4.643 so_20754161_20754198_1 . . . 94
4.644 so_23564044_23564720_2 . . . 94
4.645 so_31614816_31636049_1 . . . 94
4.646 so_19578237_19578336_2 . . . 94
4.647 so_24992091_24992436_0 . . . 94
4.648 so_33020532_33020942_2 . . . 94
4.649 so_20405569_20405891_0 . . . 94
4.650 so_33670170_33670325_0 . . . 94
4.651 so_33670170_33670255_1 . . . 94
4.652 so_39395289_39395845_1 . . . 94
4.653 so_28588827_28589112_0 . . . 94
4.654 so_28588827_28589112_1 . . . 94
4.655 so_28588827_28589107_0 . . . 94
4.656 so_42491290_42491290_2 . . . 94
4.657 so_42491290_42491502_1 . . . 94
4.658 so_17257828_17257838_2 . . . 94
4.659 so_34515544_34517233_1 . . . 94
4.660 so_47067006_47068111_3 . . . 94
4.661 so_47067006_47068111_1 . . . 94
4.662 so_29422970_29423564_1 . . . 94
4.663 so_27951344_27951801_1 . . . 94
4.664 so_40787241_40787595_1 . . . 94
4.665 so_2838107_2838701_1 . . . 94
4.666 so_18028200_18028227_0 . . . 94
4.667 so_27229578_27229578_4 . . . 94

4.668 so_27229578_27229578_2 . . . 94
4.669 so_27137362_27139030_2 . . . 94
4.670 so_15283523_15284276_5 . . . 94
4.671 so_17386722_17386795_0 . . . 94
4.672 so_43511868_43512098_1 . . . 94
4.673 so_43511868_43511923_0 . . . 94
4.674 so_28810470_28817538_6 . . . 94
4.675 so_41108304_41108347_3 . . . 94
4.676 so_15547305_15547408_0 . . . 94
4.677 so_5155820_5155820_0 . . . 94
4.678 so_19569800_19570223_6 . . . 94
4.679 so_35711301_35711773_1 . . . 94
4.680 so_41821182_41822734_1 . . . 94
4.681 so_27238670_27238859_1 . . . 94
4.682 so_13466556_14557329_1 . . . 94
4.683 so_33107787_33108109_0 . . . 94
4.684 so_39646801_39648025_1 . . . 94
4.685 so_14526942_14587960_5 . . . 94
4.686 so_27226680_27226816_0 . . . 94
4.687 so_25540033_25540631_2 . . . 94
4.688 so_1461381_1462060_5 . . . 94
4.689 so_38038122_38064418_0 . . . 94
4.690 so_15529530_15529908_1 . . . 94
4.691 so_33097257_33097478_0 . . . 94
4.692 so_6524185_6524855_3 . . . 94
4.693 so_28156381_28159822_2 . . . 94
4.694 so_39477443_39484401_0 . . . 94
4.695 so_17772552_17772777_0 . . . 94
4.696 so_39924012_39924012_0 . . . 94
4.697 so_42447811_42448309_2 . . . 94
4.698 so_5014061_5015089_3 . . . 94
4.699 so_18098788_18099299_0 . . . 94
4.700 so_21162490_21162566_0 . . . 94
4.701 so_27238670_27238859_2 . . . 94
4.702 so_29850716_29850922_0 . . . 94
4.703 so_29850716_29850921_2 . . . 94
4.704 so_25629061_25630943_2 . . . 94
4.705 so_23346749_23347213_0 . . . 94
4.706 so_18102156_18285077_1 . . . 94
4.707 so_25540033_25540631_3 . . . 94
4.708 so_7197677_7197677_0 . . . 94
4.709 so_38487640_38488280_3 . . . 94
4.710 Pointer to member conversions . . . 94
4.711 so_4665709_4665867_4 . . . 94
4.712 so_29516653_29516941_1 . . . 94
4.713 so_35153996_35154862_2 . . . 94
4.714 so_32672876_32672876_1 . . . 94
4.715 so_25119928_25120076_0 . . . 95
4.716 so_4295117_4295117_0 . . . 95
4.717 so_42894356_42894356_0 . . . 95
4.718 so_31518214_31518760_2 . . . 95

4.719 so_28156381_28159822_1 . . . 95
4.720 so_8450429_8450429_0 . . . 95
4.721 so_17479427_17504137_1 . . . 95
4.722 so_27137634_27137634_1 . . . 95
4.723 so_46750163_46750570_2 . . . 95
4.724 so_9586355_9587564_0 . . . 95
4.725 so_4272909_4273271_3 . . . 95
4.726 so_40575060_40575405_0 . . . 95
4.727 so_6942949_6946317_1 . . . 95
4.728 so_30373_293047_12 . . . 95
4.729 so_14961322_14961618_0 . . . 95
4.730 so_60000_2688631_1 . . . 95
4.731 so_60000_2688631_0 . . . 95
4.732 Boolean conversions . . . 95
4.733 so_25552822_25552822_3 . . . 95
4.734 so_33330435_33330451_0 . . . 95
4.735 so_19737929_19738076_2 . . . 95
4.736 so_16934876_16935270_0 . . . 95
4.737 so_16934876_16935023_1 . . . 95
4.738 so_32136092_32137530_6 . . . 95
4.739 so_651361_651378_1 . . . 95
4.740 so_651361_651380_3 . . . 95
4.741 so_25825818_25826267_0 . . . 95
4.742 so_9498057_9498104_0 . . . 95
4.743 so_9498057_9498127_1 . . . 95
4.744 so_23638282_23638529_2 . . . 95
4.745 so_38535292_38535667_6 . . . 95
4.746 so_16994263_16994295_0 . . . 95
4.747 so_41757634_41757976_0 . . . 95
4.748 so_15876960_15878243_6 . . . 95
4.749 so_32684369_32684432_0 . . . 95
4.750 so_11058422_11058463_11 . . . 95
4.751 so_23530206_23530445_2 . . . 95
4.752 so_2654378_2655054_2 . . . 95
4.753 so_28862203_28862503_4 . . . 95
4.754 so_25909051_25909289_1 . . . 95
4.755 so_25909051_25909939_6 . . . 95
4.756 so_25909051_25909939_4 . . . 95
4.757 so_33474487_33474550_0 . . . 95
4.758 so_18840422_18840449_1 . . . 95
4.759 so_18840422_18840464_1 . . . 95
4.760 so_18840422_18863247_1 . . . 95
4.761 so_28998613_28998685_0 . . . 95
4.762 so_6807393_6807398_2 . . . 95
4.763 so_4966002_4966034_1 . . . 95
4.764 so_22620517_22620795_1 . . . 95
4.765 so_22620517_22620776_0 . . . 95
4.766 so_40088029_40088115_0 . . . 95
4.767 so_18889968_18889969_0 . . . 95
4.768 so_40978251_40978340_1 . . . 95
4.769 so_14400385_14543166_3 . . . 95

4.770 so_15032044_15032114_7 . . . 95
4.771 so_4316316_4322041_5 . . . 95
4.772 so_34612344_34612452_1 . . . 95
4.773 so_34612344_34612453_0 . . . 95
4.774 so_32253788_32254282_2 . . . 95
4.775 so_24530868_24530906_1 . . . 95
4.776 so_13268608_13268882_0 . . . 95
4.777 so_29402367_29402590_0 . . . 95
4.778 so_28447030_28447049_1 . . . 95
4.779 so_26785827_26785980_0 . . . 95
4.780 so_25538776_25538876_0 . . . 95
4.781 so_43803368_43803501_0 . . . 95
4.782 so_43803368_43803501_1 . . . 95
4.783 so_4852249_4852249_1 . . . 95
4.784 so_22243721_22243785_4 . . . 95
4.785 so_27226680_27226680_1 . . . 95
4.786 so_28660352_28660610_2 . . . 95
4.787 so_27136524_27136708_0 . . . 95
4.788 so_27136524_27136524_0 . . . 95
4.789 so_26900158_26900182_1 . . . 95
4.790 so_33363994_33364183_0 . . . 95
4.791 so_9833790_9833830_1 . . . 95
4.792 so_9158567_9158567_0 . . . 95
4.793 so_9158567_9159641_3 . . . 95
4.794 so_29875674_29875712_0 . . . 95
4.795 so_10464291_10464390_0 . . . 95
4.796 so_1149109_1149311_1 . . . 95
4.797 so_25540033_25540631_4 . . . 95
4.798 so_25540033_25540631_7 . . . 95
4.799 so_17418584_17418710_3 . . . 95
4.800 so_17772103_17772144_0 . . . 95
4.801 so_17772103_17772117_0 . . . 95
4.802 so_25241516_25242107_7 . . . 95
4.803 so_21415606_21415632_0 . . . 95
4.804 so_21415606_21415723_0 . . . 95
4.805 so_22012911_22013065_0 . . . 95
4.806 so_17334126_17334182_0 . . . 95
4.807 Integer conversion rank . . . 95
4.808 so_36568050_36568050_0 . . . 95
4.809 so_44338559_44338825_8 . . . 95
4.810 so_40070352_40070428_3 . . . 95
4.811 so_25308761_25308909_9 . . . 95
4.812 so_25308761_25308881_12 . . . 95
4.813 so_15853353_15853557_2 . . . 95
4.814 so_19444580_19444580_0 . . . 95
4.815 so_3357614_3358012_0 . . . 95
4.816 so_30488777_30488911_9 . . . 95
4.817 so_30488777_30488911_2 . . . 95
4.818 so_30488777_30488911_10 . . . 95
4.819 so_30488777_30488911_8 . . . 95
4.820 so_30488777_30488911_4 . . . 95

4.821 so_30488777_30488911_1 . . . 95
4.822 so_30488777_30488911_3 . . . 95
4.823 so_30488777_30488911_5 . . . 95
4.824 so_30488777_30488911_0 . . . 95
4.825 so_30488777_30488911_6 . . . 95
4.826 so_30488777_30488911_7 . . . 95
4.827 so_31837603_31856601_1 . . . 95
4.828 so_4062388_4062388_0 . . . 95
4.829 so_45041045_45052333_15 . . . 95
4.830 so_11985774_12338977_12 . . . 95
4.831 so_11985774_12338977_17 . . . 95
4.832 so_11985774_12338977_9 . . . 95
4.833 so_11985774_12338977_10 . . . 95
4.834 so_11985774_12338977_13 . . . 95
4.835 so_11985774_12338977_19 . . . 95
4.836 so_11985774_12338977_11 . . . 95
4.837 so_11985774_12338977_15 . . . 95
4.838 so_11985774_12338977_18 . . . 95
4.839 so_11985774_12338977_14 . . . 95
4.840 so_11985774_12338977_16 . . . 95
4.841 so_38744365_38744467_7 . . . 95
4.842 so_36820591_36831660_9 . . . 95
4.843 so_36820591_36831660_11 . . . 95
4.844 so_36820591_36831660_6 . . . 95
4.845 so_36820591_36831660_2 . . . 95
4.846 so_36820591_36831660_3 . . . 95
4.847 so_36820591_36831660_5 . . . 95
4.848 so_36820591_36831660_10 . . . 95
4.849 so_36820591_36831660_4 . . . 95
4.850 so_36820591_36831660_7 . . . 95
4.851 so_36820591_36831660_8 . . . 95
4.852 so_46150905_46151095_1 . . . 95
4.853 so_15823217_15823253_1 . . . 95
4.854 so_20518527_20518559_6 . . . 95
4.855 so_20518527_20518559_1 . . . 95
4.856 so_37118426_37119041_8 . . . 95
4.857 so_8771409_8771433_1 . . . 95
4.858 so_8771409_8771433_2 . . . 95
4.859 so_13810222_13810408_18 . . . 95
4.860 so_4807643_4807726_7 . . . 95
4.861 so_37166131_37166556_7 . . . 95
4.862 so_34879858_34912703_9 . . . 95
4.863 so_5381882_30322474_3 . . . 95
4.864 so_46117714_46117714_0 . . . 95
4.865 so_34252403_34252511_4 . . . 95
4.866 so_34252403_34252511_3 . . . 95
4.867 so_29121176_29217925_10 . . . 95
4.868 so_27405418_27405419_6 . . . 95
4.869 so_45775764_45776027_4 . . . 95
4.870 so_45775764_45776027_2 . . . 95
4.871 so_10965325_10965371_6 . . . 95

4.872 so_14745825_14746323_3 . . . 95
4.873 so_26101257_26101357_2 . . . 95
4.874 so_26101257_26101357_3 . . . 95
4.875 so_24181172_24181220_0 . . . 95
4.876 so_34986335_34986350_1 . . . 95
4.877 so_34986335_34986391_2 . . . 95
4.878 so_37481434_37481628_8 . . . 95
4.879 so_16260033_16260033_2 . . . 95
4.880 so_31436620_31436956_8 . . . 95
4.881 so_31702505_31702699_0 . . . 95
4.882 so_31702505_31702644_4 . . . 95
4.883 so_26010806_26010806_1 . . . 95
4.884 so_26010806_26010806_2 . . . 95
4.885 so_26010806_26010806_3 . . . 95
4.886 so_46283690_46284103_2 . . . 95

5 Expressions **97**
5.1 so_18822242_18822242_0 . . . 97
5.2 so_5099769_21680645_1 . . . 97
5.3 so_38180381_38180535_0 . . . 97
5.4 so_13515434_13515524_0 . . . 97
5.5 so_7042858_7042858_0 . . . 97
5.6 so_10778627_10778627_4 . . . 97
5.7 so_39162251_39162466_1 . . . 97
5.8 so_39162251_39162585_0 . . . 97
5.9 so_26694842_26694904_1 . . . 97
5.10 so_26694842_26694842_2 . . . 97
5.11 so_43263631_43264234_3 . . . 97
5.12 so_32238645_32239072_0 . . . 97
5.13 so_35060111_35060325_0 . . . 97
5.14 so_44679200_44679576_0 . . . 97
5.15 so_23519366_23519448_0 . . . 97
5.16 so_12984861_17205195_0 . . . 97
5.17 so_18195312_18195408_3 . . . 97
5.18 so_8503229_8504241_1 . . . 97
5.19 so_675976_676008_0 . . . 97
5.20 so_17805220_17806165_1 . . . 97
5.21 so_17045661_17046802_0 . . . 97
5.22 so_6842880_6843474_3 . . . 97
5.23 so_14345548_14346544_0 . . . 97
5.24 so_13569566_28453985_3 . . . 97
5.25 so_16188263_16188846_1 . . . 97
5.26 so_9052088_9053207_0 . . . 97
5.27 so_40785566_40785774_3 . . . 97
5.28 so_17398959_17437214_0 . . . 97
5.29 so_37938701_37938946_1 . . . 97
5.30 so_40172606_40172686_3 . . . 97
5.31 so_40172606_40172686_4 . . . 97
5.32 so_17176221_17176239_1 . . . 97
5.33 so_18195715_41280273_0 . . . 97
5.34 so_36626810_36629498_0 . . . 97

5.35 so_38826007_38826971_0 . . . 97
5.36 so_42926763_42927110_0 . . . 97
5.37 so_31662792_31666090_0 . . . 97
5.38 so_19205125_19205206_0 . . . 97
5.39 so_28207856_28208832_4 . . . 97
5.40 so_17106448_17107043_1 . . . 97
5.41 so_21319413_21319414_2 . . . 97
5.42 so_23008320_23008478_0 . . . 97
5.43 so_9024826_9024826_0 . . . 97
5.44 so_9476036_9476130_0 . . . 97
5.45 so_35569139_35569331_2 . . . 97
5.46 so_35569139_35569331_1 . . . 97
5.47 so_7451207_7452397_0 . . . 97
5.48 so_3784996_24844494_0 . . . 97
5.49 so_17542124_17542153_0 . . . 97
5.50 so_23166648_23191394_0 . . . 97
5.51 so_29257855_29257928_0 . . . 97
5.52 so_10692902_10693000_0 . . . 97
5.53 so_35138864_35138909_1 . . . 97
5.54 so_12343690_12343849_1 . . . 97
5.55 so_12275248_12275310_0 . . . 97
5.56 so_46372614_46373136_0 . . . 97
5.57 so_24296571_24296605_0 . . . 97
5.58 so_24296571_24297811_1 . . . 97
5.59 so_6941733_6942028_1 . . . 97
5.60 so_29808397_29809849_9 . . . 97
5.61 so_29808397_29809849_10 . . . 97
5.62 so_20707442_20707442_0 . . . 97
5.63 so_18217369_18218389_0 . . . 97
5.64 so_26154528_26154528_0 . . . 97
5.65 so_16057690_16088912_0 . . . 97
5.66 so_26857674_26857674_6 . . . 97
5.67 so_40172392_40173491_2 . . . 97
5.68 so_1979989_1980011_0 . . . 97
5.69 so_39611311_39612307_0 . . . 97
5.70 so_28662784_28728098_5 . . . 97
5.71 so_45913725_45914957_0 . . . 97
5.72 so_32761132_32761352_1 . . . 97
5.73 so_24827592_24828578_2 . . . 97
5.74 so_3224155_3230386_0 . . . 97
5.75 so_18473496_18473628_0 . . . 97
5.76 so_17428394_17488350_0 . . . 97
5.77 so_20556587_20556770_0 . . . 97
5.78 so_16794633_16794743_4 . . . 97
5.79 so_31804946_31810042_1 . . . 97
5.80 so_40774393_40774567_1 . . . 97
5.81 so_24978819_24979030_0 . . . 97
5.82 so_7890204_7891906_0 . . . 97
5.83 so_5303324_5305219_0 . . . 97
5.84 so_17241614_17242295_0 . . . 97
5.85 so_31708731_31709049_0 . . . 97

5.86 so_16567044_16567138_0 . . . 97
5.87 so_14364552_14364612_0 . . . 97
5.88 so_46015090_46016362_0 . . . 97
5.89 so_42930598_42931969_0 . . . 97
5.90 so_27970737_27971064_0 . . . 97
5.91 so_36892827_36894485_0 . . . 97
5.92 so_28241533_28241618_0 . . . 97
5.93 so_36577718_36578069_0 . . . 97
5.94 so_15745359_15745624_0 . . . 97
5.95 so_43798210_43798800_0 . . . 97
5.96 so_42989034_42989034_0 . . . 97
5.97 so_37881240_37881475_0 . . . 97
5.98 so_17241614_17242295_1 . . . 97
5.99 so_21739621_21739994_3 . . . 97
5.100 so_21739621_21739994_1 . . . 97
5.101 so_2220230_2220254_3 . . . 97
5.102 so_10359590_10359808_3 . . . 97
5.103 so_9152798_9152814_2 . . . 97
5.104 so_13138851_13139295_1 . . . 97
5.105 so_42026327_42027557_1 . . . 97
5.106 so_11581903_11661184_6 . . . 97
5.107 so_11581903_11661184_3 . . . 97
5.108 so_11581903_11661184_1 . . . 97
5.109 so_11581903_11661184_5 . . . 97
5.110 so_11581903_11661184_2 . . . 97
5.111 so_11581903_11661184_4 . . . 97
5.112 so_24518517_24519164_2 . . . 97
5.113 so_24518517_24519164_4 . . . 97
5.114 so_24518517_24519164_5 . . . 97
5.115 so_24518517_24519164_3 . . . 97
5.116 so_24518517_24519164_6 . . . 97
5.117 so_7348673_7352048_0 . . . 97
5.118 so_26532884_26533057_1 . . . 97
5.119 so_22430998_22431501_6 . . . 97
5.120 so_9963974_9964097_1 . . . 97
5.121 so_34597738_34597738_0 . . . 97
5.122 so_31206988_31206988_3 . . . 97
5.123 so_31206988_31206988_4 . . . 97
5.124 so_31206988_31206988_2 . . . 97
5.125 so_31206988_31206988_1 . . . 97
5.126 so_17241614_17242295_13 . . . 97
5.127 so_17241614_17242295_11 . . . 97
5.128 so_17241614_17242295_15 . . . 97
5.129 so_17241614_17242295_12 . . . 97
5.130 so_17241614_17242295_14 . . . 97
5.131 so_11724592_11724592_3 . . . 97
5.132 so_11724592_11724592_1 . . . 97
5.133 so_11724592_11724592_4 . . . 97
5.134 so_11724592_11724592_2 . . . 97
5.135 so_7721184_7721284_2 . . . 98
5.136 so_18929467_18929776_0 . . . 98

5.137 so_37934918_37939803_0 . . . 98
5.138 so_46331625_46331907_1 . . . 98
5.139 so_27840695_27840837_1 . . . 98
5.140 so_32231075_32231125_1 . . . 98
5.141 so_39669783_39670474_0 . . . 98
5.142 so_35088599_35088909_0 . . . 98
5.143 so_35088599_35088845_0 . . . 98
5.144 so_26559758_30424903_1 . . . 98
5.145 so_27375117_27375280_0 . . . 98
5.146 so_14547986_14548934_1 . . . 98
5.147 so_5099769_5099769_1 . . . 98
5.148 so_21053273_21053273_2 . . . 98
5.149 so_24544525_24544722_0 . . . 98
5.150 so_24194076_24206844_6 . . . 98
5.151 so_14730534_14730687_2 . . . 98
5.152 so_43798210_43798800_1 . . . 98
5.153 so_43798210_43798620_2 . . . 98
5.154 so_20850536_20999389_7 . . . 98
5.155 so_4999241_4999832_1 . . . 98
5.156 so_25302480_25302649_7 . . . 98
5.157 so_25302480_25302649_6 . . . 98
5.158 so_25302480_25302649_4 . . . 98
5.159 so_25302480_25302649_8 . . . 98
5.160 so_24489929_24489971_4 . . . 98
5.161 so_32530475_32530759_0 . . . 98
5.162 so_32477249_32479159_1 . . . 98
5.163 so_34486886_34486987_0 . . . 98
5.164 so_34486886_34486987_1 . . . 98
5.165 so_34486886_34486987_2 . . . 98
5.166 so_34486886_34486987_3 . . . 98
5.167 so_33731158_33739185_2 . . . 98
5.168 so_33731158_33739185_1 . . . 98
5.169 so_33731158_33739185_0 . . . 98
5.170 so_33731158_33731465_0 . . . 98
5.171 so_33731158_33739185_3 . . . 98
5.172 so_40912448_40912554_1 . . . 98
5.173 so_40912448_40912554_0 . . . 98
5.174 so_4814668_4814901_1 . . . 98
5.175 so_4814668_4814901_7 . . . 98
5.176 so_4814668_4814901_0 . . . 98
5.177 so_22418979_22418996_0 . . . 98
5.178 so_22418979_22418996_1 . . . 98
5.179 so_22418979_22418996_4 . . . 98
5.180 so_38517422_38517666_0 . . . 98
5.181 so_31807052_31810394_0 . . . 98
5.182 so_39856970_39863945_4 . . . 98
5.183 so_39856970_39863945_3 . . . 98
5.184 so_17832815_17833338_2 . . . 98
5.185 so_17832815_17833338_1 . . . 98
5.186 so_17832815_17833338_0 . . . 98
5.187 so_17832815_17833338_3 . . . 98

5.188 so_17832815_17834317_0 . . . 98
5.189 so_12396132_12396208_1 . . . 98
5.190 so_3357614_3357684_3 . . . 98
5.191 so_30035547_30035924_1 . . . 98
5.192 so_30035547_30035924_3 . . . 98
5.193 so_30108274_30108422_1 . . . 98
5.194 so_30488777_30488777_4 . . . 98
5.195 so_30488777_30488777_3 . . . 98
5.196 so_30488777_30488777_1 . . . 98
5.197 so_30488777_30488777_2 . . . 98
5.198 so_30488777_30488777_0 . . . 98
5.199 so_29735103_29735125_1 . . . 98
5.200 so_35130890_35130890_0 . . . 98
5.201 so_34710492_34710865_9 . . . 98
5.202 so_34710492_34710865_12 . . . 98
5.203 so_34710492_34710865_10 . . . 98
5.204 so_34710492_34710865_13 . . . 98
5.205 so_34710492_34710865_7 . . . 98
5.206 so_34710492_34710865_11 . . . 98
5.207 so_34710492_34710865_8 . . . 98
5.208 so_13981151_13981171_1 . . . 98
5.209 so_13981151_13981171_0 . . . 98
5.210 so_42459750_42497336_4 . . . 98
5.211 so_42459750_42497336_8 . . . 98
5.212 so_42459750_42497336_3 . . . 98
5.213 so_42459750_42497336_7 . . . 98
5.214 so_42459750_42497336_6 . . . 98
5.215 so_42459750_42497336_2 . . . 98
5.216 so_42459750_42497384_0 . . . 98
5.217 so_42459750_42497336_10 . . . 98
5.218 so_42459750_42497336_11 . . . 98
5.219 so_42459750_42497336_5 . . . 98
5.220 so_42459750_42497336_9 . . . 98
5.221 so_42459750_42497336_1 . . . 98
5.222 so_31662792_31666090_3 . . . 98
5.223 so_24168460_24169125_4 . . . 98
5.224 so_24168460_24169125_6 . . . 98
5.225 so_30804659_30805815_2 . . . 98
5.226 so_27844971_27877127_0 . . . 98
5.227 so_27844971_27877127_2 . . . 98
5.228 so_38772637_38773012_1 . . . 98
5.229 so_38772637_38773012_2 . . . 98
5.230 so_36024047_36024121_3 . . . 98
5.231 so_17215484_17215694_0 . . . 98
5.232 so_15314778_15314839_0 . . . 98
5.233 so_765709_765722_3 . . . 98
5.234 so_28010565_28010631_1 . . . 98
5.235 so_28819913_28820232_1 . . . 98
5.236 so_18247919_18249553_1 . . . 98
5.237 so_18247919_18249553_0 . . . 98
5.238 so_18247919_18249553_2 . . . 98

5.239 so_18247919_18249553_3 . . . 98
5.240 so_27931157_27931379_1 . . . 98
5.241 so_721861_3739974_1 . . . 98
5.242 so_721861_3739974_2 . . . 98
5.243 so_721861_3739974_0 . . . 98
5.244 so_721861_722100_0 . . . 98
5.245 so_721861_722100_2 . . . 98
5.246 so_721861_722100_1 . . . 98
5.247 so_40329101_40329102_1 . . . 98
5.248 so_40329101_40329102_4 . . . 98
5.249 so_40329101_40329102_2 . . . 98
5.250 so_40329101_40329102_0 . . . 98
5.251 so_40329101_40329102_3 . . . 98
5.252 so_12889237_12889697_0 . . . 98
5.253 so_13254873_13254998_0 . . . 98
5.254 so_13254873_13254998_1 . . . 98
5.255 so_34252403_34252511_1 . . . 98
5.256 so_24518517_24519164_11 . . . 98
5.257 so_5800946_5800976_0 . . . 98
5.258 so_5800946_5801031_0 . . . 98
5.259 so_5800946_5800976_2 . . . 98
5.260 so_6770258_6770275_0 . . . 98
5.261 so_36182654_36183442_0 . . . 98
5.262 so_27405418_27405419_4 . . . 98
5.263 so_27405418_27405419_5 . . . 98
5.264 so_5563000_23873536_11 . . . 98
5.265 so_5563000_23873536_3 . . . 98
5.266 so_5563000_23873536_6 . . . 98
5.267 so_5563000_5563063_9 . . . 98
5.268 so_5563000_23873536_10 . . . 98
5.269 so_5563000_23873536_7 . . . 98
5.270 so_5563000_5563063_8 . . . 98
5.271 so_5563000_5563063_5 . . . 98
5.272 so_5563000_23873536_13 . . . 98
5.273 so_5563000_5563063_6 . . . 98
5.274 so_5563000_23873536_0 . . . 98
5.275 so_5563000_23873536_2 . . . 98
5.276 so_5563000_23873536_9 . . . 98
5.277 so_5563000_5563063_0 . . . 98
5.278 so_5563000_23873536_8 . . . 98
5.279 so_5563000_23873536_14 . . . 98
5.280 so_5563000_23873536_12 . . . 98
5.281 so_5563000_5563063_1 . . . 98
5.282 so_5563000_23873536_15 . . . 98
5.283 so_5563000_23873536_1 . . . 98
5.284 so_5563000_23873536_5 . . . 98
5.285 so_5563000_5563063_2 . . . 98
5.286 so_5563000_5563063_4 . . . 98
5.287 so_5563000_5563063_3 . . . 98
5.288 so_5563000_23873536_4 . . . 98
5.289 so_29402367_29402657_2 . . . 98

5.290 so_10751470_10751574_3 . . . 98
5.291 so_10751470_10751574_1 . . . 98
5.292 so_10751470_10751539_2 . . . 98
5.293 so_10751470_10751539_0 . . . 98
5.294 so_10751470_10751574_2 . . . 98
5.295 so_10751470_10751574_4 . . . 98
5.296 so_1280304_1280362_6 . . . 98
5.297 so_1280304_1280362_0 . . . 98
5.298 so_40891113_40891838_0 . . . 98
5.299 so_40891113_40891838_1 . . . 98
5.300 so_20649734_20729634_11 . . . 98
5.301 so_20649734_20729634_10 . . . 98
5.302 so_20649734_20729634_3 . . . 98
5.303 so_20649734_20729634_6 . . . 98
5.304 so_20649734_20729634_2 . . . 98
5.305 so_20649734_20729634_4 . . . 98
5.306 so_20649734_20729634_7 . . . 98
5.307 so_20649734_20729634_8 . . . 98
5.308 so_20649734_20729634_1 . . . 98
5.309 so_20649734_20729634_5 . . . 98
5.310 so_20649734_20729634_9 . . . 98
5.311 so_38737024_38738152_0 . . . 98
5.312 so_26143976_26144079_4 . . . 98
5.313 so_19471235_19471347_2 . . . 98
5.314 so_41720492_41720636_6 . . . 98
5.315 so_41720492_41720636_4 . . . 98
5.316 so_41720492_41720636_5 . . . 98
5.317 so_41720492_41720636_1 . . . 98
5.318 so_41720492_41720636_2 . . . 98
5.319 so_41720492_41720636_3 . . . 98
5.320 so_30685838_30685969_3 . . . 98
5.321 so_30685838_30685969_2 . . . 98
5.322 so_30685838_30685969_1 . . . 98
5.323 so_15662006_15662232_0 . . . 98
5.324 so_17226998_17227099_1 . . . 98
5.325 so_17226998_17227099_0 . . . 98
5.326 so_34147715_34147715_1 . . . 98
5.327 so_8566147_8566300_0 . . . 98
5.328 so_28235805_28235872_0 . . . 98
5.329 so_24067499_24067599_6 . . . 98
5.330 so_24067499_24067599_14 . . . 98
5.331 so_24067499_24067599_11 . . . 98
5.332 so_24067499_24067599_12 . . . 98
5.333 so_24067499_24067599_2 . . . 98
5.334 so_24067499_24067599_8 . . . 98
5.335 so_24067499_24067599_13 . . . 98
5.336 so_24067499_24067599_15 . . . 98
5.337 so_24067499_24067599_4 . . . 98
5.338 so_24067499_24067599_3 . . . 98
5.339 so_24067499_24067599_5 . . . 98
5.340 so_24067499_24067599_1 . . . 98

5.341 so_24067499_24067599_0 . . . 98
5.342 so_24067499_24067599_10 . . . 98
5.343 so_24067499_24067599_9 . . . 98
5.344 so_24067499_24067599_7 . . . 98
5.345 so_5780578_5780629_0 . . . 98
5.346 so_31702505_31702644_2 . . . 98
5.347 so_31702505_31702712_1 . . . 98
5.348 so_31702505_31702712_3 . . . 98
5.349 so_5692882_5692895_0 . . . 98
5.350 so_13630042_13630606_1 . . . 98
5.351 so_22143032_22143606_0 . . . 98
5.352 so_22143032_22143606_3 . . . 98
5.353 so_22143032_22143606_6 . . . 98
5.354 so_22143032_22143606_5 . . . 98
5.355 so_22143032_22143606_1 . . . 98
5.356 so_22143032_22143606_7 . . . 98
5.357 so_22143032_22143606_4 . . . 98
5.358 so_22143032_22143606_2 . . . 98
5.359 so_18679093_18679093_1 . . . 98
5.360 so_38952864_38953240_0 . . . 98
5.361 so_38952864_38952995_0 . . . 98
5.362 so_39964651_39964956_1 . . . 98
5.363 so_39964651_39964956_0 . . . 98
5.364 so_29710801_29710879_2 . . . 98
5.365 so_47336098_47336287_2 . . . 98
5.366 so_47336098_47336098_2 . . . 98
5.367 so_47336098_47336287_0 . . . 98
5.368 so_30645436_30645436_4 . . . 98
5.369 so_46283690_46284103_0 . . . 98
5.370 so_29068333_29068918_0 . . . 98
5.371 so_26939070_26939087_1 . . . 99
5.372 so_21807288_21807328_0 . . . 99
5.373 so_32154924_32155323_0 . . . 99
5.374 so_46331625_46331907_0 . . . 99
5.375 so_21053273_21053273_3 . . . 99
5.376 so_16798422_16799424_1 . . . 99
5.377 so_20242868_20243257_6 . . . 99
5.378 so_30532566_30532566_3 . . . 99
5.379 so_28482809_28483256_15 . . . 99
5.380 so_29868792_29873914_0 . . . 99
5.381 so_29868792_29873914_1 . . . 99
5.382 so_22233769_22235528_3 . . . 99
5.383 so_30329758_30330008_0 . . . 99
5.384 so_2497825_2497846_0 . . . 99
5.385 so_38751915_38751991_2 . . . 99
5.386 so_28010565_28018488_2 . . . 99
5.387 so_34389262_34389400_1 . . . 99
5.388 so_34389262_34389400_2 . . . 99
5.389 so_8802831_8803230_0 . . . 99
5.390 so_10201735_10551800_0 . . . 99
5.391 so_10201735_10551800_1 . . . 99

5.392 so_23373742_23375539_8 . . . 99
5.393 so_33673798_33673798_3 . . . 99
5.394 so_29018809_29018810_10 . . . 99
5.395 so_29018809_29018810_12 . . . 99
5.396 so_29018809_29018810_9 . . . 99
5.397 so_8366290_8366323_2 . . . 99
5.398 Primary expressions . . . 100
5.399 so_28265984_28266104_0 . . . 100
5.400 so_28176415_28176643_4 . . . 100
5.401 so_20443262_20443362_4 . . . 100
5.402 so_10004511_10004511_0 . . . 100
5.403 so_37518434_37518676_0 . . . 100
5.404 so_46600017_46601362_0 . . . 100
5.405 so_15036281_15036305_2 . . . 100
5.406 so_32519993_32521110_1 . . . 100
5.407 so_33242940_33243307_0 . . . 100
5.408 so_46527754_46527754_0 . . . 100
5.409 so_46527754_46529605_0 . . . 100
5.410 so_30293262_30293277_0 . . . 100
5.411 so_37901693_37901693_4 . . . 100
5.412 so_38267148_39675770_1 . . . 101
5.413 so_37278537_37278537_0 . . . 101
5.414 so_13993798_13995360_1 . . . 101
5.415 so_36790872_36790933_0 . . . 101
5.416 so_30422729_30443939_0 . . . 101
5.417 so_20021292_20021356_3 . . . 101
5.418 so_11092592_11093653_0 . . . 101
5.419 so_27722358_27732038_3 . . . 101
5.420 so_9018460_9018668_0 . . . 101
5.421 so_47165384_47165639_0 . . . 101
5.422 so_37278537_37278537_1 . . . 101
5.423 so_37038832_37038912_0 . . . 101
5.424 so_37038832_37039339_4 . . . 101
5.425 so_7255379_7255637_0 . . . 101
5.426 so_16524041_16524224_1 . . . 101
5.427 so_16524041_16524041_0 . . . 101
5.428 so_16524041_16524270_0 . . . 101
5.429 so_21143835_26696662_0 . . . 101
5.430 so_16524897_16524897_0 . . . 101
5.431 so_16524897_16524897_1 . . . 101
5.432 so_23136632_23136799_0 . . . 101
5.433 so_11092592_11093653_1 . . . 101
5.434 so_47165384_47165639_1 . . . 101
5.435 so_6162226_6162226_0 . . . 101
5.436 so_21143835_26696662_1 . . . 101
5.437 so_11092592_11093653_2 . . . 101
5.438 so_27722358_27732038_5 . . . 101
5.439 so_21143835_26696662_2 . . . 101
5.440 so_18145567_18146631_3 . . . 101
5.441 so_24121708_24122429_2 . . . 101
5.442 so_24116817_24116817_0 . . . 101

5.443 so_23914289_23915503_0 . . . 101
5.444 so_22817517_22817633_0 . . . 101
5.445 so_13153993_13154091_0 . . . 101
5.446 so_11509757_11510186_0 . . . 101
5.447 so_23459603_23460825_0 . . . 102
5.448 so_31761091_31761091_0 . . . 102
5.449 so_31761091_31762150_3 . . . 102
5.450 so_27722358_27722358_7 . . . 102
5.451 so_21807288_21807328_1 . . . 102
5.452 so_31206462_31206733_4 . . . 102
5.453 so_20242868_20242868_0 . . . 102
5.454 so_27840925_27881306_4 . . . 102
5.455 so_17958352_17958489_6 . . . 102
5.456 so_35834275_35834337_0 . . . 102
5.457 so_20652424_20652424_0 . . . 102
5.458 so_16567044_16567044_0 . . . 102
5.459 so_31440790_36060509_0 . . . 102
5.460 so_24299963_24303296_1 . . . 103
5.461 so_23978617_23978793_0 . . . 103
5.462 so_7119427_7120002_1 . . . 103
5.463 so_11478962_11478962_6 . . . 103
5.464 so_34935618_34936548_2 . . . 103
5.465 so_36354041_36354041_4 . . . 103
5.466 so_44280061_44280834_0 . . . 103
5.467 so_44280061_44280318_0 . . . 103
5.468 so_19060720_19061195_0 . . . 103
5.469 so_45681363_45681513_0 . . . 103
5.470 so_25321859_25321970_0 . . . 103
5.471 so_26346386_26346386_0 . . . 103
5.472 so_26744528_26744867_0 . . . 103
5.473 so_42369314_42369314_1 . . . 103
5.474 so_42369314_42369766_1 . . . 103
5.475 so_28053640_28054077_3 . . . 103
5.476 so_28053640_28054077_0 . . . 103
5.477 so_28053640_28054077_1 . . . 103
5.478 so_26199545_26199701_0 . . . 103
5.479 so_35856037_35856870_0 . . . 103
5.480 so_33988156_33988156_1 . . . 103
5.481 so_33988156_33988156_0 . . . 103
5.482 so_33834573_33848588_3 . . . 103
5.483 so_33834573_33848588_1 . . . 103
5.484 so_33834573_33848588_0 . . . 103
5.485 so_47263793_47263793_0 . . . 103
5.486 so_39536367_39536690_0 . . . 103
5.487 so_39536367_39536690_1 . . . 103
5.488 so_39536367_39536690_3 . . . 103
5.489 so_37185803_37189273_0 . . . 103
5.490 so_37185803_37189273_1 . . . 103
5.491 so_37185803_37189273_5 . . . 103
5.492 so_37185803_37189273_3 . . . 103
5.493 so_39354795_39354911_0 . . . 103

5.494 so_39354795_39354911_3 . . . 103
5.495 so_39354795_39354911_1 . . . 103
5.496 so_27159187_27160109_0 . . . 103
5.497 so_37038832_37039339_3 . . . 103
5.498 so_37038832_37039339_0 . . . 103
5.499 so_37038832_37039339_1 . . . 103
5.500 so_27840695_27840837_4 . . . 103
5.501 so_27840695_27840837_2 . . . 103
5.502 so_12656548_12656790_0 . . . 103
5.503 so_28778671_28778769_0 . . . 103
5.504 so_44280061_44280834_1 . . . 103
5.505 so_44280061_44280834_4 . . . 103
5.506 so_44280061_44280834_2 . . . 103
5.507 so_13863219_13882851_0 . . . 103
5.508 so_39832455_39832859_0 . . . 103
5.509 so_39832455_39832859_2 . . . 103
5.510 so_15945298_15945444_3 . . . 104
5.511 so_24221061_24221355_2 . . . 104
5.512 so_47521694_47529137_2 . . . 104
5.513 so_15945298_15945444_4 . . . 104
5.514 so_26168164_26178478_1 . . . 104
5.515 so_26168164_26178478_0 . . . 104
5.516 so_33074110_33074895_1 . . . 104
5.517 so_44485610_44485702_0 . . . 104
5.518 so_44485610_44485702_1 . . . 104
5.519 so_18673004_20734293_0 . . . 104
5.520 so_4661875_4661946_0 . . . 104
5.521 so_44946408_44946408_0 . . . 104
5.522 so_22232164_22232165_1 . . . 104
5.523 so_22232164_22232164_0 . . . 104
5.524 so_24499004_24499084_1 . . . 104
5.525 so_24273136_24273136_1 . . . 104
5.526 so_24237729_24252540_1 . . . 104
5.527 so_37809623_37811379_0 . . . 104
5.528 so_5849059_5849247_1 . . . 104
5.529 so_16921247_16922211_1 . . . 104
5.530 so_14801591_14804071_0 . . . 104
5.531 so_17188485_17190112_0 . . . 104
5.532 so_22998364_23002021_8 . . . 104
5.533 so_22998364_23002021_10 . . . 104
5.534 so_22998364_23002021_7 . . . 104
5.535 so_22998364_23002021_9 . . . 104
5.536 so_22998364_23002021_6 . . . 104
5.537 so_8481997_8482018_0 . . . 104
5.538 so_11933999_11934046_0 . . . 104
5.539 so_11933999_12086873_0 . . . 104
5.540 so_18755787_18755787_0 . . . 104
5.541 so_18755787_18755787_4 . . . 104
5.542 so_18755787_18755787_2 . . . 104
5.543 so_18755787_18755912_0 . . . 104
5.544 so_18755787_18755787_1 . . . 104

5.545 so_18755787_18755787_3 . . . 104
5.546 so_47521694_47529137_1 . . . 104
5.547 so_29038703_29038703_1 . . . 104
5.548 so_29038703_29038703_2 . . . 104
5.549 so_29038703_29038703_4 . . . 104
5.550 so_29038703_29038703_0 . . . 104
5.551 so_29038703_29038703_3 . . . 104
5.552 so_22719066_22719985_1 . . . 104
5.553 so_22719066_22719985_2 . . . 104
5.554 so_22719066_22719985_4 . . . 104
5.555 so_22719066_22719985_3 . . . 104
5.556 so_22719066_22719985_5 . . . 104
5.557 so_22719066_22719985_0 . . . 104
5.558 so_18432260_18432260_0 . . . 104
5.559 so_29558677_29581627_0 . . . 104
5.560 so_34667733_34667761_1 . . . 104
5.561 so_37232481_37232747_0 . . . 104
5.562 so_37232481_37232971_0 . . . 104
5.563 so_33170759_33170934_1 . . . 104
5.564 so_28366572_28371192_1 . . . 104
5.565 so_33074110_33074895_2 . . . 104
5.566 so_33074110_33074901_0 . . . 104
5.567 so_28498080_28498182_0 . . . 104
5.568 so_19149150_19149568_0 . . . 104
5.569 so_11534265_11534379_0 . . . 104
5.570 so_11534265_11534408_0 . . . 104
5.571 so_16357681_16357708_0 . . . 104
5.572 so_40736293_40736333_0 . . . 104
5.573 so_24210455_24210746_0 . . . 104
5.574 so_27814714_27814829_0 . . . 104
5.575 so_24499004_24499084_2 . . . 104
5.576 so_46421139_46428235_0 . . . 104
5.577 so_46421139_46428235_1 . . . 104
5.578 so_24273136_24273136_2 . . . 104
5.579 so_24237729_24252540_2 . . . 104
5.580 so_34596254_34596319_0 . . . 104
5.581 so_17822131_17822241_1 . . . 104
5.582 so_47518542_47518542_2 . . . 104
5.583 so_47518542_47518542_1 . . . 104
5.584 so_15279547_15279657_0 . . . 104
5.585 so_15279547_15279659_0 . . . 104
5.586 so_15279547_15279658_1 . . . 104
5.587 so_24839157_24839696_1 . . . 104
5.588 so_45056703_45066215_0 . . . 104
5.589 so_26834655_26834832_2 . . . 104
5.590 so_26834655_26834832_4 . . . 104
5.591 so_26834655_26834832_1 . . . 104
5.592 so_26834655_26834832_5 . . . 104
5.593 so_26834655_26834832_0 . . . 104
5.594 so_26834655_26834832_3 . . . 104
5.595 so_26834655_26834736_0 . . . 104

5.596 so_26834655_26834832_9 . . . 104
5.597 so_18949577_18950081_0 . . . 104
5.598 so_18661149_18661198_0 . . . 104
5.599 so_7851427_7851565_1 . . . 104
5.600 so_23834845_23835015_0 . . . 104
5.601 so_12731215_12731215_0 . . . 104
5.602 so_32986193_32986347_4 . . . 104
5.603 so_32986193_32986339_0 . . . 104
5.604 so_32986193_32986347_1 . . . 104
5.605 so_32986193_32986347_0 . . . 104
5.606 so_32986193_32986347_2 . . . 104
5.607 so_32986193_32986347_5 . . . 104
5.608 so_32986193_32986347_3 . . . 104
5.609 so_30240131_30241944_0 . . . 104
5.610 so_8391058_8391983_1 . . . 104
5.611 so_26187711_26187743_0 . . . 104
5.612 so_10480359_10480474_0 . . . 104
5.613 so_32308742_32308828_0 . . . 104
5.614 so_39933440_39933440_0 . . . 104
5.615 so_8539924_8539959_1 . . . 104
5.616 so_37481767_37482103_5 . . . 104
5.617 so_37481767_37482103_6 . . . 104
5.618 so_37481767_37482103_1 . . . 104
5.619 so_37481767_37481993_0 . . . 104
5.620 so_37481767_37482103_4 . . . 104
5.621 so_37481767_37482103_3 . . . 104
5.622 so_37481767_37482103_7 . . . 104
5.623 so_28263207_28263286_0 . . . 104
5.624 so_27593001_27593064_0 . . . 104
5.625 so_32898066_32898379_0 . . . 104
5.626 so_43970960_43971134_0 . . . 104
5.627 so_10479564_10479674_0 . . . 104
5.628 so_7951377_20825525_0 . . . 104
5.629 so_29915596_29916211_0 . . . 104
5.630 so_41534031_41537651_0 . . . 105
5.631 so_27610469_27610512_0 . . . 105
5.632 so_27610469_27610552_2 . . . 105
5.633 so_26692637_26692775_5 . . . 105
5.634 so_28389792_28390022_0 . . . 105
5.635 so_22998364_23002021_1 . . . 105
5.636 so_21961474_21961474_0 . . . 105
5.637 so_31648458_31649056_3 . . . 105
5.638 so_31648458_31649056_0 . . . 105
5.639 so_29038703_29038870_0 . . . 105
5.640 so_14048919_14049072_0 . . . 105
5.641 so_42240849_42241015_0 . . . 105
5.642 so_19149150_19149568_1 . . . 105
5.643 so_17233547_17233649_0 . . . 105
5.644 so_11866573_11867022_0 . . . 105
5.645 so_32299171_32299536_3 . . . 105
5.646 so_24680321_24683065_0 . . . 105

5.647 so_6880111_6880148_0 . . . 105
5.648 so_17683591_17683799_1 . . . 105
5.649 so_17683591_17683799_0 . . . 105
5.650 so_24839157_24839696_3 . . . 105
5.651 so_28916109_28917094_0 . . . 105
5.652 so_26834655_26834832_7 . . . 105
5.653 so_26834655_26834832_10 . . . 105
5.654 so_2611357_2611616_0 . . . 105
5.655 so_34545551_34545653_0 . . . 105
5.656 so_30240851_30241368_0 . . . 105
5.657 so_30240131_30241944_1 . . . 105
5.658 so_8391058_8391983_2 . . . 105
5.659 so_27058847_27059232_0 . . . 105
5.660 so_16571125_16571198_1 . . . 105
5.661 so_7951377_20825525_1 . . . 105
5.662 so_15618415_15622599_0 . . . 105
5.663 so_25827766_25827822_0 . . . 105
5.664 so_5501959_5502084_0 . . . 105
5.665 so_26667515_26667544_0 . . . 105
5.666 so_17771671_17772810_0 . . . 106
5.667 so_41720989_41721337_0 . . . 106
5.668 so_27339123_27339356_0 . . . 106
5.669 so_34234384_34234505_0 . . . 106
5.670 so_8513050_8513174_0 . . . 106
5.671 so_14450608_14450664_0 . . . 106
5.672 so_28746744_28746827_0 . . . 106
5.673 so_39146874_39147737_0 . . . 106
5.674 so_2935201_28767216_0 . . . 106
5.675 so_2935201_28767216_1 . . . 106
5.676 so_2935201_2935230_0 . . . 106
5.677 so_25355327_25355423_0 . . . 106
5.678 so_18889028_18889029_0 . . . 106
5.679 so_15657011_15674512_0 . . . 106
5.680 so_25546249_25546491_0 . . . 106
5.681 so_18673004_20734293_1 . . . 106
5.682 so_26220448_26220448_0 . . . 106
5.683 so_4691056_4691274_2 . . . 106
5.684 so_26626158_26626573_0 . . . 106
5.685 so_24390911_24392099_0 . . . 106
5.686 so_17822131_17822241_2 . . . 106
5.687 so_27857253_27857253_1 . . . 106
5.688 so_24215683_24215683_0 . . . 106
5.689 so_37898525_37898704_0 . . . 106
5.690 so_26834655_26834832_11 . . . 106
5.691 so_26834655_26834832_8 . . . 106
5.692 so_11468414_11468500_0 . . . 106
5.693 so_27970737_27970833_0 . . . 106
5.694 so_4855219_4868535_1 . . . 106
5.695 so_28429352_28429453_0 . . . 106
5.696 so_8026170_8026489_0 . . . 106
5.697 so_26829385_26829520_1 . . . 106

5.698 so_26748736_26748772_0 . . . 106
5.699 so_21000882_21004439_0 . . . 106
5.700 so_15861168_15861184_0 . . . 106
5.701 so_28263102_28263163_0 . . . 106
5.702 so_37493769_37493769_0 . . . 106
5.703 so_43213997_43213997_0 . . . 106
5.704 so_43213997_43238095_0 . . . 106
5.705 so_8539924_8539959_0 . . . 106
5.706 so_27593001_27593064_1 . . . 106
5.707 so_19961873_19962132_0 . . . 106
5.708 so_27989031_27989291_6 . . . 106
5.709 so_31602985_31603101_1 . . . 106
5.710 so_33299031_33299167_0 . . . 106
5.711 so_33299031_33299163_1 . . . 106
5.712 so_32211029_32211148_0 . . . 106
5.713 so_32211029_32211207_0 . . . 106
5.714 so_7951377_20825525_2 . . . 106
5.715 so_4726768_4730655_0 . . . 106
5.716 so_7210818_7210880_0 . . . 106
5.717 so_28141403_28141551_0 . . . 106
5.718 so_16680992_16681036_0 . . . 106
5.719 so_16323032_16323119_0 . . . 107
5.720 so_22998364_23002021_2 . . . 107
5.721 so_7764564_7764680_1 . . . 107
5.722 so_43827651_43827695_0 . . . 107
5.723 so_37144183_37144183_0 . . . 107
5.724 so_33285103_33289327_0 . . . 107
5.725 so_4589056_4593286_0 . . . 107
5.726 so_32097759_32098850_0 . . . 107
5.727 so_24839157_24841967_0 . . . 107
5.728 so_2122282_2122404_0 . . . 107
5.729 so_42528917_42559940_0 . . . 107
5.730 so_8391058_8391983_3 . . . 107
5.731 so_12905539_12905607_0 . . . 107
5.732 so_11128020_11128519_0 . . . 107
5.733 so_13844148_13844505_0 . . . 107
5.734 so_31390941_31391619_0 . . . 107
5.735 so_39766467_39770060_0 . . . 107
5.736 so_10160489_10160597_0 . . . 107
5.737 so_42088015_42089831_2 . . . 107
5.738 so_36479812_36480125_0 . . . 107
5.739 so_16323032_16323119_1 . . . 107
5.740 so_12989774_12989774_0 . . . 107
5.741 so_37753998_37754133_0 . . . 107
5.742 so_17872224_17872224_0 . . . 107
5.743 so_7214623_7215486_0 . . . 108
5.744 so_43827651_43827695_2 . . . 108
5.745 so_21443023_21443273_0 . . . 108
5.746 so_25934368_25934438_1 . . . 108
5.747 so_14639401_14644511_0 . . . 108
5.748 so_34926326_34926996_0 . . . 108

5.749 so_15612513_15612655_2 . . . 108
5.750 so_19344955_19344955_0 . . . 108
5.751 so_6111617_6111617_0 . . . 108
5.752 so_7214623_7215486_1 . . . 108
5.753 so_8666492_8666594_1 . . . 108
5.754 so_9199744_9201213_1 . . . 108
5.755 so_33285103_33285103_3 . . . 108
5.756 so_33285103_33285103_4 . . . 108
5.757 so_33285103_33289327_2 . . . 108
5.758 so_5470174_5470174_0 . . . 108
5.759 so_19344955_19345096_2 . . . 108
5.760 so_20361865_20361865_0 . . . 108
5.761 so_40099376_40099376_0 . . . 108
5.762 so_31919260_31919871_5 . . . 108
5.763 so_31919260_31919654_0 . . . 108
5.764 so_42046021_42046360_0 . . . 108
5.765 so_32894748_32895274_4 . . . 108
5.766 so_31040915_32895827_0 . . . 108
5.767 so_21390608_21588021_1 . . . 108
5.768 so_19018100_19018329_0 . . . 108
5.769 so_38750777_38750777_3 . . . 108
5.770 so_38750777_38750777_4 . . . 108
5.771 so_38750777_38750777_0 . . . 108
5.772 so_40406560_40406880_1 . . . 108
5.773 so_34323489_34324971_0 . . . 108
5.774 so_34323489_34324971_3 . . . 108
5.775 so_34323489_34324971_2 . . . 108
5.776 so_9199744_9201213_0 . . . 108
5.777 so_42582531_42582531_0 . . . 108
5.778 so_6181464_6181507_0 . . . 108
5.779 so_21443023_21443273_1 . . . 108
5.780 so_26040452_26040651_3 . . . 108
5.781 so_26726920_26727161_0 . . . 108
5.782 so_40242505_40243927_1 . . . 108
5.783 so_18949577_18950081_1 . . . 108
5.784 so_13743727_13743727_2 . . . 108
5.785 so_33849718_33849718_0 . . . 108
5.786 so_33849718_33849718_2 . . . 108
5.787 so_46724260_46725055_0 . . . 108
5.788 so_46724260_46725795_0 . . . 108
5.789 so_46724260_46725055_2 . . . 108
5.790 so_46724260_46725795_2 . . . 108
5.791 so_15612513_15612583_0 . . . 108
5.792 so_15612513_15612655_1 . . . 108
5.793 so_28141403_28141551_1 . . . 108
5.794 so_28141403_28141551_3 . . . 108
5.795 so_19018100_19018329_1 . . . 109
5.796 so_43827651_43827695_1 . . . 109
5.797 so_40406560_40406880_3 . . . 109
5.798 so_42582531_42600027_1 . . . 109
5.799 so_42610429_42611583_0 . . . 109

5.800 so_14270941_14271402_2 . . . 109
5.801 so_43119643_43120923_0 . . . 109
5.802 so_33285103_33289327_1 . . . 109
5.803 so_28763375_28763570_0 . . . 109
5.804 so_34926326_34926996_1 . . . 109
5.805 so_15612513_15612583_1 . . . 109
5.806 so_28141403_28141551_4 . . . 109
5.807 so_36071961_36071997_0 . . . 109
5.808 so_43467095_43467268_0 . . . 109
5.809 so_20361865_20362378_0 . . . 109
5.810 so_7741652_7741652_0 . . . 109
5.811 so_19676473_19676545_0 . . . 110
5.812 so_12216793_12216934_0 . . . 110
5.813 so_12718423_12718425_0 . . . 110
5.814 so_22998364_23002021_3 . . . 110
5.815 so_33170759_33170934_0 . . . 110
5.816 so_31485041_31485150_0 . . . 110
5.817 so_19963041_19976680_0 . . . 110
5.818 so_46421139_46428235_2 . . . 110
5.819 so_33755358_33755546_0 . . . 110
5.820 so_23918872_23919639_0 . . . 110
5.821 so_24839157_24839696_5 . . . 110
5.822 so_45056703_45057356_0 . . . 110
5.823 so_43361845_43364126_4 . . . 110
5.824 so_43361845_43364126_5 . . . 110
5.825 so_18949577_18950081_2 . . . 110
5.826 so_21118609_21118609_0 . . . 110
5.827 so_27954634_27954691_0 . . . 110
5.828 so_29729671_29729671_2 . . . 110
5.829 so_19996134_19996354_0 . . . 110
5.830 so_32894748_32894748_0 . . . 110
5.831 so_32894748_32895274_3 . . . 110
5.832 so_15612513_15612655_0 . . . 110
5.833 so_12520611_12520816_0 . . . 110
5.834 so_21390608_21392743_6 . . . 110
5.835 so_21390608_21392743_2 . . . 110
5.836 so_26667515_26667544_1 . . . 110
5.837 so_33436150_33436340_0 . . . 110
5.838 so_33436150_33436457_0 . . . 110
5.839 so_40522392_40522392_0 . . . 110
5.840 so_46421139_46428235_3 . . . 110
5.841 so_31318762_31319260_0 . . . 110
5.842 so_21443023_21443273_3 . . . 110
5.843 so_21118609_21118609_1 . . . 110
5.844 so_29729671_29729671_1 . . . 110
5.845 so_13468989_13468989_0 . . . 110
5.846 so_21390608_21392743_7 . . . 110
5.847 so_21390608_21392743_3 . . . 110
5.848 so_22998364_23002021_4 . . . 110
5.849 so_46900655_46907206_0 . . . 110
5.850 so_40131096_40131922_0 . . . 110

5.851 so_31318762_31319260_1 . . . 110
5.852 so_28763375_28763570_1 . . . 110
5.853 so_33854388_33854742_1 . . . 110
5.854 so_24839157_24839696_6 . . . 110
5.855 so_42528917_42559940_2 . . . 110
5.856 so_18949577_18950081_3 . . . 110
5.857 so_24388577_24388917_1 . . . 110
5.858 so_21118609_21118609_2 . . . 110
5.859 so_29729671_29729981_0 . . . 110
5.860 so_32894748_32895274_0 . . . 110
5.861 so_13468989_13471654_0 . . . 110
5.862 so_31040915_32895827_1 . . . 110
5.863 so_33854388_33854742_2 . . . 111
5.864 so_13743727_13743727_3 . . . 111
5.865 so_32894748_32895274_1 . . . 111
5.866 so_31040915_32895827_2 . . . 111
5.867 so_40331193_40331211_1 . . . 111
5.868 so_26168164_26170459_1 . . . 111
5.869 so_11669292_11669652_0 . . . 111
5.870 so_11255027_11255089_1 . . . 111
5.871 so_38024024_38024297_0 . . . 111
5.872 so_35190585_35190714_1 . . . 111
5.873 so_40223413_40223929_0 . . . 111
5.874 so_26687426_26687426_1 . . . 111
5.875 so_20750669_20750820_0 . . . 111
5.876 so_38541354_38541830_1 . . . 111
5.877 so_38541354_38541830_8 . . . 111
5.878 so_8849823_8849823_1 . . . 111
5.879 so_17493360_17497984_0 . . . 111
5.880 so_32911729_32911729_0 . . . 111
5.881 so_6186877_6186877_0 . . . 111
5.882 so_12718423_12718425_1 . . . 111
5.883 so_19921954_19922161_0 . . . 111
5.884 so_25895430_25895624_0 . . . 111
5.885 so_19996134_19996354_1 . . . 111
5.886 so_15423695_15423723_0 . . . 111
5.887 so_12463269_12463335_0 . . . 111
5.888 so_21443023_21443273_2 . . . 111
5.889 so_23534488_23534770_0 . . . 111
5.890 so_31795823_31796009_0 . . . 111
5.891 so_24816557_24816772_0 . . . 112
5.892 Postfix expressions . . . 112
5.893 so_47307601_47307768_1 . . . 112
5.894 so_22330845_22331715_1 . . . 112
5.895 so_46429692_46429926_0 . . . 112
5.896 so_46910844_46910844_0 . . . 112
5.897 so_21010685_21011586_0 . . . 112
5.898 so_40590216_40590301_2 . . . 112
5.899 so_40590216_40620526_1 . . . 112
5.900 so_46613077_46613188_0 . . . 112
5.901 so_29327339_29327400_0 . . . 112

5.902 so_25987790_25987942_1 . . . 112
5.903 so_37051199_37052522_1 . . . 112
5.904 so_22272920_22272974_0 . . . 112
5.905 so_30724010_30724060_1 . . . 112
5.906 so_32077611_32077759_0 . . . 112
5.907 so_36757924_36758232_1 . . . 112
5.908 so_36757924_36758232_0 . . . 112
5.909 so_16568389_16568708_0 . . . 112
5.910 so_18145567_18146631_0 . . . 112
5.911 so_34688755_34689060_0 . . . 112
5.912 so_36559112_36559135_1 . . . 112
5.913 so_34982357_34982780_0 . . . 112
5.914 so_8111357_8111377_2 . . . 112
5.915 so_30514796_30514936_0 . . . 112
5.916 so_36834260_36834652_0 . . . 112
5.917 so_22237123_22237155_0 . . . 112
5.918 so_34274861_34274996_0 . . . 112
5.919 so_23872692_23872852_0 . . . 112
5.920 so_2310838_31213185_0 . . . 112
5.921 so_27162576_27162630_0 . . . 112
5.922 so_32906247_32906349_0 . . . 112
5.923 so_32906247_32906350_0 . . . 112
5.924 so_18727022_18727216_2 . . . 112
5.925 so_3612554_3612966_0 . . . 112
5.926 so_33197793_33197955_1 . . . 112
5.927 so_32069280_32069562_0 . . . 112
5.928 so_32069280_32069494_0 . . . 112
5.929 so_35798287_35798369_0 . . . 112
5.930 so_721861_3739974_5 . . . 112
5.931 so_28920949_28921140_0 . . . 112
5.932 so_988158_988220_1 . . . 112
5.933 so_45765452_45765660_1 . . . 112
5.934 so_29243116_29244316_0 . . . 112
5.935 so_12336511_14385036_0 . . . 112
5.936 so_42203474_42203903_0 . . . 112
5.937 so_28976327_28976386_0 . . . 112
5.938 so_5460562_5460977_0 . . . 112
5.939 so_29242409_29242623_0 . . . 112
5.940 so_18781840_18782612_1 . . . 112
5.941 so_18781840_18782612_0 . . . 112
5.942 so_42750701_42750701_0 . . . 112
5.943 so_31206988_33218377_1 . . . 112
5.944 so_31206988_31206988_5 . . . 112
5.945 so_20378447_20378499_0 . . . 112
5.946 so_20378447_20378553_2 . . . 112
5.947 so_23000629_23000771_0 . . . 112
5.948 so_35606590_35606629_0 . . . 112
5.949 so_24800112_24800112_0 . . . 112
5.950 so_40902397_40902500_1 . . . 112
5.951 so_33161003_33161401_1 . . . 112
5.952 so_33161003_33161103_0 . . . 112

5.953 so_5073350_5073366_0 . . . 112
5.954 so_6734649_6734649_0 . . . 113
5.955 so_24800112_24800112_1 . . . 113
5.956 so_23895081_23895527_6 . . . 113
5.957 so_31448845_31449668_1 . . . 113
5.958 so_7170394_7170548_1 . . . 113
5.959 so_45265690_45372988_1 . . . 113
5.960 so_43774223_43774686_1 . . . 113
5.961 so_39319320_39319650_1 . . . 113
5.962 so_38180381_38180535_1 . . . 113
5.963 so_38180381_38180544_0 . . . 113
5.964 so_16253248_16253369_0 . . . 113
5.965 so_38837592_38837592_2 . . . 113
5.966 so_28129777_28130423_5 . . . 113
5.967 so_23703601_23703753_3 . . . 113
5.968 so_12140820_12140936_0 . . . 113
5.969 so_22782166_22782166_0 . . . 113
5.970 so_23960436_23960657_1 . . . 113
5.971 so_23960436_23960657_4 . . . 113
5.972 so_23960436_23960657_3 . . . 113
5.973 so_23960436_23960657_2 . . . 113
5.974 so_23518993_23543949_1 . . . 113
5.975 so_2068022_2072973_0 . . . 113
5.976 so_2474018_2474021_0 . . . 113
5.977 so_17480440_17480596_1 . . . 113
5.978 so_41262456_41262532_0 . . . 113
5.979 so_4855219_4868535_0 . . . 113
5.980 so_22494302_22494503_0 . . . 113
5.981 so_22494302_22494302_0 . . . 113
5.982 so_31869026_31869097_0 . . . 113
5.983 so_695734_695744_0 . . . 113
5.984 so_28449304_28449489_1 . . . 113
5.985 so_8948443_8949164_0 . . . 113
5.986 so_5968273_5968583_1 . . . 113
5.987 so_26918903_26919328_0 . . . 113
5.988 so_26918903_26919072_0 . . . 113
5.989 so_21143835_21148117_1 . . . 113
5.990 so_29116751_29136042_3 . . . 113
5.991 so_23519074_23520450_1 . . . 113
5.992 so_34588930_34591141_3 . . . 113
5.993 so_7761527_7761536_1 . . . 113
5.994 so_39162251_39162585_1 . . . 113
5.995 so_24020788_24021170_0 . . . 113
5.996 so_42804326_42805525_0 . . . 113
5.997 so_20707442_20708771_0 . . . 113
5.998 so_20707442_20707442_1 . . . 113
5.999 so_2944794_2945818_1 . . . 113
5.1000 so_27840925_27881306_3 . . . 113
5.1001 so_3158999_3430035_2 . . . 113
5.1002 so_39824814_39824879_2 . . . 113
5.1003 so_39824814_39824814_0 . . . 113

5.1004 so_39349005_39349199_1 . . . 113
5.1005 so_30347703_30347759_0 . . . 113
5.1006 so_7170394_7170548_2 . . . 113
5.1007 so_37155404_37156393_0 . . . 113
5.1008 so_16492736_16492888_0 . . . 113
5.1009 so_27195825_27195825_0 . . . 113
5.1010 so_12558065_12558136_0 . . . 113
5.1011 so_24289069_24289162_1 . . . 113
5.1012 so_35042548_35043929_1 . . . 113
5.1013 so_31090216_31090325_0 . . . 113
5.1014 so_9241255_9241363_3 . . . 113
5.1015 so_16071154_16071220_0 . . . 113
5.1016 so_43185293_43186035_0 . . . 113
5.1017 so_40066406_40066451_1 . . . 113
5.1018 so_39413568_39413568_0 . . . 113
5.1019 so_7628333_7628345_0 . . . 113
5.1020 so_15680489_15680881_3 . . . 113
5.1021 so_15680489_15680881_2 . . . 113
5.1022 so_23107507_23107796_0 . . . 113
5.1023 so_24840033_24841031_1 . . . 113
5.1024 so_24840033_24841031_2 . . . 113
5.1025 so_40696973_40697195_0 . . . 113
5.1026 so_26249588_26251595_0 . . . 113
5.1027 so_7229234_7229381_1 . . . 113
5.1028 so_36992039_36992143_0 . . . 113
5.1029 so_14737680_14737791_1 . . . 113
5.1030 so_35060111_35061625_1 . . . 113
5.1031 so_35060111_35061625_2 . . . 113
5.1032 so_35060111_35061625_0 . . . 113
5.1033 so_42066811_42067082_10 . . . 113
5.1034 so_25598299_25598419_0 . . . 113
5.1035 so_24435954_24436482_0 . . . 113
5.1036 so_10048130_10048295_0 . . . 113
5.1037 so_27157204_27157694_0 . . . 113
5.1038 so_43129165_43129165_0 . . . 114
5.1039 so_11053982_11054068_1 . . . 114
5.1040 so_8245027_8245027_0 . . . 114
5.1041 so_4937454_4938713_0 . . . 114
5.1042 so_11985774_12338977_0 . . . 114
5.1043 so_3605296_3605601_0 . . . 114
5.1044 so_7997539_7998091_0 . . . 114
5.1045 so_14432539_14432703_1 . . . 114
5.1046 so_14432539_14432703_0 . . . 114
5.1047 so_23773374_23773551_0 . . . 114
5.1048 so_28110699_28117353_0 . . . 114
5.1049 so_1657883_16338804_1 . . . 114
5.1050 so_26559758_30424903_2 . . . 114
5.1051 so_28097564_28097654_1 . . . 114
5.1052 so_18742323_18742440_0 . . . 114
5.1053 so_18742323_18742418_0 . . . 114
5.1054 so_17845005_17845106_0 . . . 114

5.1055 so_21362308_21365420_0 . . . 114
5.1056 so_15026671_15026735_0 . . . 114
5.1057 so_11985774_12338977_2 . . . 114
5.1058 so_11985774_12338977_1 . . . 114
5.1059 so_34064593_34064763_0 . . . 114
5.1060 so_6546669_6546669_0 . . . 114
5.1061 so_25796126_25796713_0 . . . 114
5.1062 so_17357898_17357898_0 . . . 114
5.1063 so_31343107_31343351_0 . . . 114
5.1064 so_11016262_11016416_1 . . . 114
5.1065 so_300986_302169_0 . . . 114
5.1066 so_30490592_30490746_0 . . . 114
5.1067 so_39211727_39212207_0 . . . 114
5.1068 so_4898618_4899199_0 . . . 114
5.1069 so_4898618_4899051_1 . . . 114
5.1070 so_10083844_10083921_0 . . . 114
5.1071 so_24720028_24720119_1 . . . 114
5.1072 so_37344370_37345867_1 . . . 114
5.1073 so_833882_833929_0 . . . 114
5.1074 so_40827303_40827376_5 . . . 114
5.1075 so_38837592_38837662_0 . . . 114
5.1076 so_15166907_15168020_1 . . . 114
5.1077 so_23703601_23703716_0 . . . 114
5.1078 so_31508658_31508658_0 . . . 114
5.1079 so_43263631_43264232_0 . . . 114
5.1080 so_39238411_39238441_0 . . . 114
5.1081 so_36992039_36992152_1 . . . 114
5.1082 so_36992039_36992250_0 . . . 114
5.1083 so_18516510_18516712_1 . . . 114
5.1084 so_2934904_2934965_0 . . . 114
5.1085 so_33846577_33846669_0 . . . 114
5.1086 so_39442136_39442223_0 . . . 114
5.1087 so_33015748_33015952_2 . . . 114
5.1088 so_24173491_24173556_1 . . . 114
5.1089 so_18460912_18461000_0 . . . 114
5.1090 so_4518598_4518646_2 . . . 114
5.1091 so_30521292_30521292_0 . . . 114
5.1092 so_34673460_34673460_2 . . . 114
5.1093 so_31125447_31125846_1 . . . 114
5.1094 so_40533815_40534093_1 . . . 114
5.1095 so_8467914_8467973_0 . . . 114
5.1096 so_11581903_11661602_0 . . . 114
5.1097 so_16346068_16347248_0 . . . 114
5.1098 so_27840925_27881306_2 . . . 114
5.1099 so_22952480_23140119_4 . . . 114
5.1100 so_22952480_23140119_5 . . . 114
5.1101 so_11911056_11911111_0 . . . 114
5.1102 so_14012309_14012334_0 . . . 114
5.1103 so_16775034_16775133_0 . . . 114
5.1104 so_37043401_37055970_1 . . . 114
5.1105 so_40533815_40534093_4 . . . 114

5.1106 so_40533815_40534093_3 . . . 114
5.1107 so_11911056_11911111_4 . . . 114
5.1108 so_11911056_11911111_5 . . . 114
5.1109 so_26432941_26433065_0 . . . 115
5.1110 so_1051379_30295099_1 . . . 115
5.1111 so_28283215_28283405_3 . . . 115
5.1112 so_40765232_40765401_1 . . . 115
5.1113 so_2963396_2963451_1 . . . 115
5.1114 so_28199177_28201770_1 . . . 115
5.1115 so_10533500_10533894_0 . . . 115
5.1116 so_44563748_44564204_0 . . . 115
5.1117 so_1652396_1652404_1 . . . 115
5.1118 so_1652396_1652404_0 . . . 115
5.1119 so_26480413_26481317_0 . . . 115
5.1120 so_3484371_3487099_0 . . . 115
5.1121 so_24303735_24312655_1 . . . 115
5.1122 so_5048921_5049667_0 . . . 115
5.1123 so_44322372_44323802_0 . . . 115
5.1124 so_38977808_38977808_0 . . . 115
5.1125 so_13906959_13907057_0 . . . 115
5.1126 so_41003088_41003364_4 . . . 115
5.1127 so_27135677_27135791_1 . . . 115
5.1128 so_23202296_23202480_0 . . . 115
5.1129 so_11581903_11661602_3 . . . 115
5.1130 so_35431651_35433452_1 . . . 115
5.1131 so_39300653_43576047_6 . . . 115
5.1132 so_37255200_37255383_1 . . . 115
5.1133 so_31935470_31935708_0 . . . 115
5.1134 so_31935470_31935708_2 . . . 115
5.1135 so_27443532_27443602_0 . . . 115
5.1136 so_24623071_24623317_5 . . . 115
5.1137 so_26323471_26323825_0 . . . 115
5.1138 so_8289950_8292768_1 . . . 115
5.1139 so_21554577_21554578_4 . . . 115
5.1140 so_21973064_21973403_0 . . . 115
5.1141 so_31772588_31773124_1 . . . 115
5.1142 so_262932_263016_0 . . . 115
5.1143 so_29269025_29269162_0 . . . 115
5.1144 so_20638968_20639122_0 . . . 115
5.1145 so_20478193_20478796_0 . . . 115
5.1146 so_25499183_25499402_0 . . . 115
5.1147 so_9522965_9523417_0 . . . 115
5.1148 so_33079486_33079574_2 . . . 115
5.1149 so_18948966_18949137_0 . . . 115
5.1150 so_16690957_16691215_0 . . . 115
5.1151 so_30603512_30603673_1 . . . 115
5.1152 so_27463842_27463881_0 . . . 115
5.1153 so_6798474_6799244_0 . . . 115
5.1154 so_8067568_8068246_1 . . . 115
5.1155 so_32544354_32550704_0 . . . 115
5.1156 so_8024790_8025043_1 . . . 115

5.1157 so_27668269_27668438_5 . . . 115
5.1158 so_27668269_27668438_1 . . . 115
5.1159 so_2293796_2293875_0 . . . 115
5.1160 so_25835324_25835372_0 . . . 115
5.1161 so_37701321_37707777_1 . . . 115
5.1162 so_25109064_25410342_0 . . . 115
5.1163 so_39279074_39279847_0 . . . 115
5.1164 so_39300653_43576047_7 . . . 115
5.1165 so_25856451_25856716_0 . . . 115
5.1166 so_43798210_43798620_4 . . . 115
5.1167 so_27960122_27960408_0 . . . 115
5.1168 so_28688522_28688663_0 . . . 115
5.1169 so_906734_906794_0 . . . 115
5.1170 so_13527568_13527994_0 . . . 115
5.1171 so_2144012_2144012_0 . . . 115
5.1172 so_18948966_18949137_1 . . . 115
5.1173 so_28199177_28201770_3 . . . 115
5.1174 so_35233624_35234036_0 . . . 115
5.1175 so_24385410_24385410_1 . . . 115
5.1176 so_34478737_34480149_0 . . . 115
5.1177 so_34707571_34708035_0 . . . 115
5.1178 so_41491414_41492306_0 . . . 115
5.1179 so_39300653_43576047_8 . . . 115
5.1180 so_28688522_28688663_1 . . . 115
5.1181 so_33161003_33161401_2 . . . 115
5.1182 so_8982575_8982727_0 . . . 115
5.1183 so_46429692_46429926_2 . . . 115
5.1184 so_24000710_24000744_1 . . . 115
5.1185 so_24000710_24000744_2 . . . 115
5.1186 so_2693199_2693199_2 . . . 115
5.1187 so_26307698_26307877_0 . . . 115
5.1188 so_24000244_24000244_0 . . . 115
5.1189 so_30206864_30207180_0 . . . 115
5.1190 so_33966809_33967781_0 . . . 115
5.1191 so_32668828_32668873_0 . . . 115
5.1192 so_3546340_3546582_2 . . . 115
5.1193 so_30206864_30207180_2 . . . 115
5.1194 so_30206864_30207180_1 . . . 115
5.1195 so_29791808_29792779_0 . . . 116
5.1196 so_38398475_38401144_1 . . . 116
5.1197 so_38398475_38401144_0 . . . 116
5.1198 so_5248877_5248877_0 . . . 116
5.1199 so_3257393_3257590_4 . . . 116
5.1200 so_3257393_3257590_3 . . . 116
5.1201 so_28482809_28483256_2 . . . 116
5.1202 so_28482809_28483256_1 . . . 116
5.1203 so_39557407_39557478_1 . . . 116
5.1204 so_39557407_39557478_3 . . . 116
5.1205 so_21554577_21554578_1 . . . 116
5.1206 so_21554577_21554578_2 . . . 116
5.1207 so_21554577_21554578_0 . . . 116

5.1208 so_24221061_24221355_5 . . . 116
5.1209 so_24221061_24221355_4 . . . 116
5.1210 so_27158812_27158813_3 . . . 116
5.1211 so_26561296_26561296_0 . . . 116
5.1212 so_43016421_43016421_0 . . . 116
5.1213 so_4263796_4263851_0 . . . 116
5.1214 so_6996600_6996740_1 . . . 116
5.1215 so_23516402_23516402_0 . . . 116
5.1216 so_23516402_23516402_1 . . . 116
5.1217 so_9489609_9489679_1 . . . 116
5.1218 so_2470255_2470834_0 . . . 116
5.1219 so_44699176_44699334_0 . . . 116
5.1220 so_6683411_6683622_0 . . . 116
5.1221 so_43007483_43007483_0 . . . 116
5.1222 so_13295978_13296423_0 . . . 116
5.1223 so_3257393_3257590_1 . . . 116
5.1224 so_2474018_2474021_2 . . . 116
5.1225 so_42570752_42570876_0 . . . 116
5.1226 so_14217746_14218744_1 . . . 116
5.1227 so_25844637_25844808_1 . . . 116
5.1228 so_21447243_21447284_0 . . . 116
5.1229 so_28482809_28483256_0 . . . 116
5.1230 so_26435084_26435208_1 . . . 116
5.1231 so_14156640_14157336_0 . . . 116
5.1232 so_7049528_7049548_3 . . . 116
5.1233 so_35756466_35756466_0 . . . 116
5.1234 so_24358606_24358853_0 . . . 116
5.1235 so_28643625_28645988_0 . . . 116
5.1236 so_27936398_27938165_0 . . . 116
5.1237 so_24637514_24637563_0 . . . 116
5.1238 so_27345204_27345362_0 . . . 116
5.1239 so_2220230_2221470_0 . . . 116
5.1240 so_26561296_26561532_0 . . . 116
5.1241 so_42444262_42444707_0 . . . 116
5.1242 so_33834573_33848588_5 . . . 116
5.1243 so_41411009_41412882_2 . . . 116
5.1244 so_41411009_41412882_1 . . . 116
5.1245 so_41411009_41412882_3 . . . 116
5.1246 so_7787059_7787059_0 . . . 116
5.1247 so_11581903_11661602_1 . . . 116
5.1248 so_13162465_13162653_0 . . . 116
5.1249 so_26950959_26951975_0 . . . 116
5.1250 so_16542026_16543835_0 . . . 116
5.1251 so_16542026_16542381_0 . . . 116
5.1252 so_16542026_16543835_1 . . . 116
5.1253 so_16542026_16542026_0 . . . 116
5.1254 so_27345204_27345362_1 . . . 116
5.1255 so_16346068_16347248_1 . . . 116
5.1256 so_16346068_16347248_2 . . . 116
5.1257 so_28482809_28483254_3 . . . 116
5.1258 so_28482809_28483254_4 . . . 116

5.1259 so_42037687_42037687_0 . . . 116
5.1260 so_26082959_26083391_1 . . . 116
5.1261 so_3983536_3987551_2 . . . 116
5.1262 so_3983536_3987551_0 . . . 116
5.1263 so_3983536_3987551_1 . . . 116
5.1264 so_44626415_44720780_1 . . . 117
5.1265 so_19729591_19729681_0 . . . 117
5.1266 so_11089986_11090033_0 . . . 117
5.1267 so_30558980_30559173_0 . . . 117
5.1268 so_25329576_25329691_1 . . . 117
5.1269 so_25329576_25329691_7 . . . 117
5.1270 so_27021465_27021971_1 . . . 117
5.1271 so_35136597_35136597_0 . . . 117
5.1272 so_24965255_24965318_0 . . . 117
5.1273 so_44977326_44977601_0 . . . 117
5.1274 so_17280755_17280812_1 . . . 117
5.1275 so_17280755_17280812_0 . . . 117
5.1276 so_19069681_19069981_3 . . . 117
5.1277 so_18258745_18258857_1 . . . 117
5.1278 so_17400137_17403467_2 . . . 117
5.1279 so_2236482_2237654_0 . . . 117
5.1280 so_598148_599564_3 . . . 117
5.1281 so_598148_599564_2 . . . 117
5.1282 so_21351799_21351799_3 . . . 117
5.1283 so_14798953_14799057_0 . . . 117
5.1284 so_43263631_43264234_4 . . . 117
5.1285 so_13744507_13745174_3 . . . 117
5.1286 so_30166983_30167116_0 . . . 117
5.1287 so_29825352_30343753_0 . . . 117
5.1288 so_29825352_29825553_0 . . . 117
5.1289 so_9781437_9781778_0 . . . 117
5.1290 so_46671624_46671799_0 . . . 117
5.1291 so_18819570_18819629_0 . . . 117
5.1292 so_25654339_25656964_3 . . . 117
5.1293 so_31624692_31624740_0 . . . 117
5.1294 so_30166983_30167116_1 . . . 117
5.1295 so_3450420_3450554_0 . . . 117
5.1296 so_24298821_24299050_0 . . . 117
5.1297 so_27836451_27836567_0 . . . 117
5.1298 so_39538625_39538687_1 . . . 117
5.1299 so_11301514_11301514_0 . . . 117
5.1300 so_4644753_4644783_0 . . . 117
5.1301 so_45881213_45881386_0 . . . 117
5.1302 so_4227328_4227378_0 . . . 117
5.1303 so_13783312_13783510_1 . . . 117
5.1304 so_17129728_17129766_0 . . . 117
5.1305 so_23480947_23481011_0 . . . 117
5.1306 so_4203479_4203479_0 . . . 117
5.1307 so_11301514_11301514_1 . . . 117
5.1308 so_34901154_34901154_5 . . . 117
5.1309 so_34901154_34902396_0 . . . 117

5.1310 so_44587814_44588251_1 . . . 117
5.1311 so_33847651_33847735_1 . . . 117
5.1312 so_11301514_11301637_0 . . . 117
5.1313 so_11301514_11301514_2 . . . 117
5.1314 so_41108304_41108347_1 . . . 117
5.1315 so_5155820_5155876_0 . . . 117
5.1316 so_11301514_11301514_3 . . . 117
5.1317 so_11301514_11301637_1 . . . 117
5.1318 so_25102816_25103554_9 . . . 117
5.1319 so_24298821_24298847_1 . . . 117
5.1320 so_24298821_24299050_1 . . . 117
5.1321 so_44330470_44330926_0 . . . 117
5.1322 so_33847651_33847735_2 . . . 117
5.1323 so_6927895_6928085_1 . . . 117
5.1324 so_25838258_25838501_1 . . . 117
5.1325 so_11301514_11301514_4 . . . 117
5.1326 so_4644753_4644783_2 . . . 117
5.1327 so_25102816_25103554_8 . . . 117
5.1328 so_25102816_25103008_2 . . . 117
5.1329 so_25102816_25103008_1 . . . 117
5.1330 so_25102816_25103554_6 . . . 117
5.1331 so_24298821_24299050_2 . . . 118
5.1332 so_23945043_23945179_0 . . . 118
5.1333 so_40202953_40203019_0 . . . 118
5.1334 so_24920218_24920218_1 . . . 118
5.1335 so_46769981_46771164_0 . . . 118
5.1336 so_25229748_25229749_0 . . . 118
5.1337 so_25838258_25838501_2 . . . 118
5.1338 so_4203479_4203479_1 . . . 118
5.1339 so_11301514_11301514_5 . . . 118
5.1340 so_4644753_4644783_4 . . . 118
5.1341 so_4227328_4227378_2 . . . 118
5.1342 so_13783312_13783510_2 . . . 118
5.1343 so_33592986_33593183_0 . . . 118
5.1344 so_9475267_9475371_0 . . . 118
5.1345 so_6927895_6928085_3 . . . 118
5.1346 so_25610023_25610023_0 . . . 118
5.1347 so_11301514_11301514_6 . . . 118
5.1348 so_22875320_22897555_3 . . . 118
5.1349 so_23075200_23077680_0 . . . 118
5.1350 so_33592986_33593183_2 . . . 118
5.1351 so_33592986_33593183_1 . . . 118
5.1352 so_33592986_33593183_3 . . . 118
5.1353 so_7207384_7207506_0 . . . 118
5.1354 so_27214556_27214615_1 . . . 118
5.1355 so_27214556_27214615_2 . . . 118
5.1356 so_27214556_27214615_0 . . . 118
5.1357 so_19090880_19091328_0 . . . 118
5.1358 so_38035722_38035722_0 . . . 118
5.1359 so_27042002_27042233_0 . . . 118
5.1360 so_27042002_27042233_1 . . . 118

5.1361 so_27042002_27042248_0 . . . 118
5.1362 so_27042002_27042248_1 . . . 118
5.1363 so_27042002_27042233_2 . . . 118
5.1364 so_6927895_6928085_4 . . . 118
5.1365 so_20015309_20015438_0 . . . 118
5.1366 so_11301514_11301514_9 . . . 118
5.1367 so_11301514_11301514_7 . . . 118
5.1368 so_11301514_11301514_8 . . . 118
5.1369 so_25102816_25103554_3 . . . 118
5.1370 so_25102816_25103554_2 . . . 118
5.1371 so_25102816_25103554_1 . . . 118
5.1372 so_26384746_26384927_2 . . . 118
5.1373 so_26384746_26384927_0 . . . 118
5.1374 so_26384746_26384927_1 . . . 118
5.1375 so_13783312_13783510_3 . . . 118
5.1376 so_5715865_5716020_0 . . . 118
5.1377 so_27878186_27878913_0 . . . 118
5.1378 so_10400890_10401083_1 . . . 118
5.1379 so_10400890_10401083_2 . . . 118
5.1380 so_10400890_10401083_0 . . . 118
5.1381 so_25589158_25589207_0 . . . 118
5.1382 so_26807743_26807798_0 . . . 118
5.1383 so_33592986_33593183_5 . . . 118
5.1384 so_16476613_16478061_1 . . . 118
5.1385 so_27214556_27214615_4 . . . 118
5.1386 so_37913950_37913980_0 . . . 118
5.1387 so_6927895_6928085_5 . . . 118
5.1388 so_30258621_30258715_0 . . . 118
5.1389 so_1276847_1276872_0 . . . 118
5.1390 so_1276847_1276873_0 . . . 118
5.1391 so_11301514_11301514_11 . . . 118
5.1392 so_25102816_25103554_12 . . . 118
5.1393 so_47310700_47310910_2 . . . 118
5.1394 so_4194191_4194298_0 . . . 119
5.1395 so_16052163_16052476_0 . . . 119
5.1396 so_7024818_7024859_0 . . . 119
5.1397 so_7024818_7024843_0 . . . 119
5.1398 so_3890047_3890097_0 . . . 119
5.1399 so_1687860_1688142_1 . . . 119
5.1400 so_13203589_13203609_0 . . . 119
5.1401 so_37307587_37308508_0 . . . 119
5.1402 so_9489609_9489820_2 . . . 119
5.1403 so_38796125_38796304_0 . . . 119
5.1404 so_17010884_17010977_0 . . . 119
5.1405 so_36702773_36708283_1 . . . 119
5.1406 so_37699188_37701796_0 . . . 119
5.1407 so_31683281_31683447_0 . . . 119
5.1408 so_24277379_24277450_1 . . . 119
5.1409 so_6795890_28120744_0 . . . 119
5.1410 so_33574112_33574286_0 . . . 119
5.1411 so_37567362_37567498_0 . . . 119

5.1412 so_28482809_28483256_7 . . . 119
5.1413 so_28482809_28483256_5 . . . 119
5.1414 so_24077253_24077577_0 . . . 119
5.1415 so_24077253_24077619_2 . . . 119
5.1416 so_24077253_24077606_6 . . . 119
5.1417 so_24077253_24077619_1 . . . 119
5.1418 so_18147864_18149769_0 . . . 119
5.1419 so_8921061_8922558_0 . . . 119
5.1420 so_31708731_31709049_2 . . . 119
5.1421 so_15898121_15898163_0 . . . 119
5.1422 so_6028427_6028427_0 . . . 119
5.1423 so_28053640_28054077_4 . . . 119
5.1424 so_17010884_17010977_1 . . . 119
5.1425 so_37699188_37701796_1 . . . 119
5.1426 so_31683281_31683447_1 . . . 119
5.1427 so_33402414_33403801_0 . . . 119
5.1428 so_27037290_27038950_0 . . . 119
5.1429 so_18147864_18149769_1 . . . 119
5.1430 so_8921061_8922558_1 . . . 119
5.1431 so_35088599_35088909_1 . . . 119
5.1432 so_6028427_6039817_0 . . . 119
5.1433 so_37061481_37061668_0 . . . 119
5.1434 so_5151875_5151914_0 . . . 119
5.1435 so_5151875_5151971_0 . . . 119
5.1436 so_14815823_14816004_0 . . . 119
5.1437 so_37412265_37412452_0 . . . 119
5.1438 so_37412265_37412396_0 . . . 119
5.1439 so_40774393_40774567_4 . . . 119
5.1440 so_39561074_39578354_0 . . . 119
5.1441 so_41087252_41087317_0 . . . 119
5.1442 so_40774393_40774567_3 . . . 119
5.1443 so_8921061_8922558_2 . . . 120
5.1444 so_27935026_30153809_1 . . . 120
5.1445 so_38988852_38989036_1 . . . 120
5.1446 so_31577762_31578597_2 . . . 120
5.1447 so_31125447_31125846_0 . . . 120
5.1448 so_20242868_20243257_5 . . . 120
5.1449 so_24770653_24770692_0 . . . 120
5.1450 so_33949332_33950387_0 . . . 120
5.1451 so_29116751_29116887_4 . . . 120
5.1452 so_5060879_5061030_2 . . . 120
5.1453 so_7501146_7501218_0 . . . 120
5.1454 so_27913603_27913893_0 . . . 120
5.1455 so_24920218_24920218_0 . . . 120
5.1456 so_15529530_15529908_0 . . . 120
5.1457 so_15529530_15529908_2 . . . 120
5.1458 so_28810470_28817538_0 . . . 120
5.1459 so_27412514_27412702_0 . . . 120
5.1460 so_34328457_34328800_0 . . . 120
5.1461 so_16567005_16567025_0 . . . 120
5.1462 so_42447811_42448309_4 . . . 120

5.1463 so_33597281_33598117_1 . . . 120
5.1464 so_6845695_6845752_2 . . . 120
5.1465 so_36119709_36119915_0 . . . 120
5.1466 so_47310700_47310910_1 . . . 120
5.1467 so_14952504_14954413_0 . . . 120
5.1468 so_46309148_46309425_0 . . . 120
5.1469 so_9819298_9820448_0 . . . 120
5.1470 so_26372292_26372399_0 . . . 120
5.1471 so_45861479_45865802_0 . . . 120
5.1472 so_33619134_33621562_0 . . . 120
5.1473 so_8464400_8585258_1 . . . 120
5.1474 so_37013517_37013647_1 . . . 120
5.1475 so_18538427_18538525_0 . . . 120
5.1476 so_3747066_3751742_0 . . . 120
5.1477 so_6414981_6415158_0 . . . 120
5.1478 so_6414981_6415142_7 . . . 120
5.1479 so_26357392_26357392_0 . . . 120
5.1480 so_28810470_28817538_5 . . . 120
5.1481 so_35425488_35427581_1 . . . 120
5.1482 so_32713083_32713134_1 . . . 120
5.1483 so_6376580_6376875_0 . . . 120
5.1484 so_34491892_34492744_0 . . . 120
5.1485 so_24385410_24385410_0 . . . 120
5.1486 so_3625410_3628383_0 . . . 120
5.1487 so_29162620_29168643_10 . . . 120
5.1488 so_22844449_22844672_1 . . . 120
5.1489 so_6845695_6845752_0 . . . 120
5.1490 so_1243962_1244346_0 . . . 120
5.1491 so_35431651_35433452_2 . . . 120
5.1492 so_37255200_37255383_2 . . . 120
5.1493 so_13466556_14557329_2 . . . 120
5.1494 so_31428980_31430017_0 . . . 120
5.1495 so_26323471_26323825_1 . . . 120
5.1496 so_25745759_25758367_0 . . . 120
5.1497 so_34829036_34829346_0 . . . 120
5.1498 so_868306_868470_0 . . . 120
5.1499 so_27695552_27695928_0 . . . 120
5.1500 so_32427433_32428111_2 . . . 120
5.1501 so_9819298_9820448_1 . . . 120
5.1502 so_29269025_29269162_1 . . . 120
5.1503 so_38487640_38488280_0 . . . 120
5.1504 so_37013517_37013647_2 . . . 120
5.1505 so_29180947_29181049_5 . . . 120
5.1506 so_43511868_43512098_0 . . . 120
5.1507 so_45134220_45134428_0 . . . 120
5.1508 so_26115775_26117470_2 . . . 121
5.1509 so_26939070_26939087_0 . . . 121
5.1510 so_29180947_29181049_6 . . . 121
5.1511 so_32828288_32828393_0 . . . 121
5.1512 so_4031228_4031487_1 . . . 121
5.1513 so_11079380_11079807_0 . . . 121

5.1514 so_11794318_11795127_2 . . . 121
5.1515 so_27080061_27080417_3 . . . 121
5.1516 so_6845695_6845752_1 . . . 121
5.1517 so_33323932_33324009_3 . . . 121
5.1518 so_34288844_34289000_0 . . . 121
5.1519 so_30532566_30532566_1 . . . 121
5.1520 so_689677_7160837_0 . . . 121
5.1521 so_39628799_39629142_0 . . . 121
5.1522 so_17881866_17881940_0 . . . 121
5.1523 so_35551197_35553018_2 . . . 121
5.1524 so_26559758_30424903_3 . . . 121
5.1525 so_22371122_22372936_2 . . . 121
5.1526 so_25395534_25395582_0 . . . 121
5.1527 so_1863069_1863093_6 . . . 121
5.1528 so_26559758_30424903_4 . . . 121
5.1529 so_17142167_31182873_2 . . . 121
5.1530 so_39303483_39303512_0 . . . 121
5.1531 so_18195715_41280273_3 . . . 121
5.1532 so_45484581_45484581_0 . . . 121
5.1533 so_18752430_18752683_1 . . . 121
5.1534 so_41276869_41277144_1 . . . 121
5.1535 so_3892924_3892965_0 . . . 121
5.1536 so_18303682_18303778_0 . . . 121
5.1537 so_24566576_24639661_0 . . . 121
5.1538 so_24566576_24634262_2 . . . 121
5.1539 so_18195312_18195408_1 . . . 121
5.1540 so_32924363_32924957_0 . . . 121
5.1541 so_33607284_33607345_1 . . . 121
5.1542 so_6974292_6974415_0 . . . 121
5.1543 so_30852922_30853213_2 . . . 121
5.1544 so_37631962_37632676_1 . . . 121
5.1545 so_27935026_30153809_2 . . . 121
5.1546 so_45484581_45484581_1 . . . 121
5.1547 so_24610483_24610483_0 . . . 121
5.1548 so_24610483_24610483_1 . . . 121
5.1549 so_7676817_7676940_2 . . . 121
5.1550 so_30122500_30122881_4 . . . 121
5.1551 so_7295632_7295699_2 . . . 121
5.1552 so_6845695_6845752_3 . . . 121
5.1553 so_21037316_21038239_0 . . . 121
5.1554 so_33607809_33607809_0 . . . 121
5.1555 so_33607809_33608071_0 . . . 121
5.1556 so_14821846_14821846_2 . . . 121
5.1557 so_14821846_14821846_0 . . . 121
5.1558 so_28477393_28477495_0 . . . 121
5.1559 so_33812998_33813557_0 . . . 121
5.1560 so_11551853_11551924_1 . . . 121
5.1561 so_27632176_27632572_4 . . . 121
5.1562 so_27632176_27632572_5 . . . 121
5.1563 so_24566576_24639661_1 . . . 121
5.1564 so_8679534_8679688_0 . . . 121

5.1565 so_8679534_8679907_1 . . . 121
5.1566 so_6025466_6025575_0 . . . 121
5.1567 so_22475615_22477750_0 . . . 121
5.1568 so_44840835_44842528_1 . . . 121
5.1569 so_6322949_6322985_0 . . . 121
5.1570 so_3747066_3748987_0 . . . 121
5.1571 so_19741843_19743719_0 . . . 121
5.1572 so_38038122_38064418_1 . . . 121
5.1573 so_33077242_33077797_1 . . . 121
5.1574 so_41921408_41921517_0 . . . 121
5.1575 so_6927895_6928085_8 . . . 121
5.1576 so_31412007_31412110_0 . . . 121
5.1577 so_7484913_7484984_0 . . . 121
5.1578 so_46205151_46205275_0 . . . 121
5.1579 so_20263888_20264283_2 . . . 121
5.1580 so_20263888_20264283_0 . . . 121
5.1581 so_20263888_20264213_1 . . . 121
5.1582 so_20263888_20264213_0 . . . 121
5.1583 so_28928590_28930230_0 . . . 121
5.1584 so_26733828_26741609_2 . . . 121
5.1585 so_12956399_12956638_0 . . . 121
5.1586 so_42098826_42099342_0 . . . 121
5.1587 so_37481434_37481695_0 . . . 121
5.1588 so_5967914_5968004_0 . . . 121
5.1589 so_5967914_5967966_0 . . . 121
5.1590 so_10278199_10278199_0 . . . 121
5.1591 so_32589328_32589328_0 . . . 121
5.1592 so_12239783_12239892_1 . . . 121
5.1593 so_1872571_1872860_1 . . . 121
5.1594 so_41171260_41171504_0 . . . 121
5.1595 so_14914940_14914996_2 . . . 121
5.1596 so_24566576_24639661_2 . . . 121
5.1597 so_4295117_4295117_1 . . . 121
5.1598 so_16059116_16067411_0 . . . 121
5.1599 so_41347826_41347931_0 . . . 121
5.1600 so_37662100_37662548_0 . . . 121
5.1601 so_17479427_17504137_3 . . . 121
5.1602 so_4272909_4273271_2 . . . 121
5.1603 so_4272909_4273271_1 . . . 121
5.1604 so_4272909_4272909_0 . . . 121
5.1605 so_4272909_4273271_0 . . . 121
5.1606 so_32013915_32014392_0 . . . 121
5.1607 so_32672876_32673117_0 . . . 121
5.1608 so_46818864_46828101_0 . . . 121
5.1609 so_9142034_9142530_0 . . . 121
5.1610 so_14961322_14961618_1 . . . 121
5.1611 so_60000_2688631_2 . . . 121
5.1612 so_27137362_27139030_3 . . . 122
5.1613 so_15283523_15284276_6 . . . 122
5.1614 so_18929225_18929285_5 . . . 122
5.1615 so_18929225_18929285_4 . . . 122

5.1616 so_18929225_18929278_1 . . . 122
5.1617 so_5057614_8465952_2 . . . 122
5.1618 so_45273291_45283790_1 . . . 122
5.1619 so_26917194_26917282_0 . . . 122
5.1620 so_573294_573334_0 . . . 122
5.1621 so_28810470_28817538_7 . . . 122
5.1622 so_14730896_14732445_1 . . . 122
5.1623 so_2614392_2614478_0 . . . 122
5.1624 so_7917045_7917190_1 . . . 122
5.1625 so_22690248_22690339_1 . . . 122
5.1626 so_22690248_22690339_2 . . . 122
5.1627 so_32487579_32487636_0 . . . 122
5.1628 so_16563114_16563422_1 . . . 122
5.1629 so_22371122_22372936_3 . . . 122
5.1630 so_2613387_2613393_0 . . . 122
5.1631 so_30745684_30746964_1 . . . 122
5.1632 so_30745684_30745684_0 . . . 122
5.1633 so_42492332_42492369_0 . . . 122
5.1634 so_25683034_25683932_1 . . . 122
5.1635 so_6845695_6845752_4 . . . 122
5.1636 so_8466452_8466452_0 . . . 122
5.1637 so_41821182_41822734_0 . . . 122
5.1638 so_24471039_24471606_1 . . . 122
5.1639 so_39137335_39137439_4 . . . 122
5.1640 so_13466556_14557329_3 . . . 122
5.1641 so_36962583_36962635_1 . . . 122
5.1642 so_22404418_22404604_0 . . . 122
5.1643 so_23519074_23520450_0 . . . 122
5.1644 so_46021568_46027039_0 . . . 122
5.1645 so_27229578_27229578_5 . . . 122
5.1646 so_19476818_19477123_0 . . . 122
5.1647 so_26559758_30424903_5 . . . 122
5.1648 so_14623266_14623461_0 . . . 122
5.1649 so_14623266_14623381_0 . . . 122
5.1650 so_19569800_19570223_0 . . . 122
5.1651 so_26793072_26793233_0 . . . 122
5.1652 so_2205353_2205421_0 . . . 122
5.1653 so_39401136_39401437_1 . . . 122
5.1654 so_42420116_42420229_0 . . . 122
5.1655 so_3574840_3575162_0 . . . 122
5.1656 so_10618473_10618866_1 . . . 122
5.1657 so_3820363_3820388_0 . . . 122
5.1658 so_15795400_15800039_0 . . . 122
5.1659 so_18867500_18870431_0 . . . 122
5.1660 so_18867500_18867601_0 . . . 122
5.1661 so_7507818_7508245_0 . . . 122
5.1662 so_14302168_14302247_1 . . . 122
5.1663 so_25585950_25586098_0 . . . 122
5.1664 so_46735324_46736040_0 . . . 122
5.1665 so_43012458_43012458_0 . . . 122
5.1666 so_31882308_31882751_8 . . . 122

5.1667 so_41957525_41957929_0 . . . 122
5.1668 so_24646084_24646084_1 . . . 122
5.1669 so_12228562_12228616_2 . . . 122
5.1670 so_2140319_2140413_1 . . . 122
5.1671 so_1229751_1551599_5 . . . 122
5.1672 so_24048701_24088981_0 . . . 122
5.1673 so_23638947_23639209_1 . . . 122
5.1674 so_11116209_11116280_0 . . . 122
5.1675 so_28979126_29893260_0 . . . 122
5.1676 so_22107838_22108026_7 . . . 122
5.1677 so_19569800_19570223_1 . . . 122
5.1678 so_8459424_8459520_0 . . . 122
5.1679 so_606728_609661_0 . . . 122
5.1680 so_21574449_21574902_0 . . . 122
5.1681 so_47597333_47598075_1 . . . 122
5.1682 so_39319862_39332627_0 . . . 122
5.1683 so_22624472_22624510_1 . . . 122
5.1684 so_21249978_21250110_0 . . . 122
5.1685 so_25265824_25265824_0 . . . 122
5.1686 so_34737737_34748943_1 . . . 122
5.1687 so_8459219_8459219_0 . . . 122
5.1688 so_8459219_8459240_0 . . . 122
5.1689 so_30628742_30631419_4 . . . 122
5.1690 so_30628742_30631419_0 . . . 122
5.1691 so_23638947_23639209_0 . . . 122
5.1692 so_23638947_23639329_0 . . . 122
5.1693 so_37506417_37506786_0 . . . 122
5.1694 so_28979126_29893260_1 . . . 122
5.1695 so_22107838_22108026_8 . . . 122
5.1696 so_19792536_19793083_1 . . . 122
5.1697 so_1994186_1994384_1 . . . 122
5.1698 so_25650592_25656153_3 . . . 122
5.1699 so_7917045_7917190_0 . . . 122
5.1700 so_24212792_24212940_2 . . . 122
5.1701 so_19569800_19570223_2 . . . 122
5.1702 so_8459424_8459520_1 . . . 122
5.1703 so_606728_606889_5 . . . 122
5.1704 so_606728_609661_1 . . . 122
5.1705 so_606728_609661_2 . . . 122
5.1706 so_3327631_3327659_0 . . . 122
5.1707 so_22624472_22624510_2 . . . 122
5.1708 so_21249978_21250110_1 . . . 122
5.1709 so_25265824_25265824_1 . . . 122
5.1710 so_4561210_4561509_0 . . . 122
5.1711 so_3468958_3469264_0 . . . 122
5.1712 so_30628742_30631419_1 . . . 122
5.1713 so_3567905_3568187_0 . . . 122
5.1714 so_3567905_17881962_0 . . . 122
5.1715 so_10164764_10169130_0 . . . 122
5.1716 so_32134719_32135786_0 . . . 122
5.1717 so_25401538_25402274_1 . . . 123

5.1718 so_31479552_31479828_0 . . . 123
5.1719 so_7170394_7170548_0 . . . 123
5.1720 so_35551197_35553018_3 . . . 123
5.1721 so_43774223_43774686_0 . . . 123
5.1722 so_8943782_8956525_0 . . . 123
5.1723 so_340413_341330_0 . . . 123
5.1724 so_27208502_27208546_2 . . . 123
5.1725 so_27208502_27208585_0 . . . 123
5.1726 so_39319745_39320518_0 . . . 123
5.1727 so_19569800_19570223_3 . . . 123
5.1728 so_1304736_1305081_0 . . . 123
5.1729 so_47597333_47598075_0 . . . 123
5.1730 so_4944436_4944451_0 . . . 123
5.1731 so_11238653_11240372_0 . . . 123
5.1732 so_32206720_32252780_1 . . . 123
5.1733 so_37606648_37606800_0 . . . 123
5.1734 so_37606648_37606711_0 . . . 123
5.1735 so_31661791_31661888_0 . . . 123
5.1736 so_2140319_2148068_0 . . . 123
5.1737 so_2140319_2140413_3 . . . 123
5.1738 so_9888102_9888219_0 . . . 123
5.1739 so_8464974_8465066_0 . . . 123
5.1740 so_8464974_8464974_0 . . . 123
5.1741 so_8555175_8555343_0 . . . 123
5.1742 so_1863069_1863093_0 . . . 123
5.1743 so_25131019_25131113_0 . . . 123
5.1744 so_13995748_13995920_0 . . . 123
5.1745 so_42212372_42213393_0 . . . 123
5.1746 so_42212372_42213393_1 . . . 123
5.1747 so_46961876_46962807_0 . . . 123
5.1748 so_8629835_8629868_2 . . . 123
5.1749 so_8629835_8629868_0 . . . 123
5.1750 so_8629835_8629868_1 . . . 123
5.1751 so_7748070_7748110_0 . . . 123
5.1752 so_6538632_6538690_0 . . . 123
5.1753 so_9234217_9234299_2 . . . 123
5.1754 so_9234217_9234287_0 . . . 123
5.1755 so_14272578_14272578_1 . . . 123
5.1756 so_24646084_24646191_1 . . . 123
5.1757 so_27137362_27137362_0 . . . 123
5.1758 so_27137362_27139030_1 . . . 123
5.1759 so_8365143_8365381_0 . . . 123
5.1760 so_19476818_19477123_2 . . . 123
5.1761 so_15283523_15284276_1 . . . 123
5.1762 so_29452124_29453071_0 . . . 123
5.1763 so_11219159_11219159_0 . . . 123
5.1764 so_21573712_21574273_0 . . . 123
5.1765 so_5057614_8465952_1 . . . 123
5.1766 so_5057614_8465952_5 . . . 123
5.1767 so_7291874_7291895_2 . . . 123
5.1768 so_29015708_29016046_2 . . . 123

5.1769 so_7507818_7508245_1 . . . 123
5.1770 so_24626972_24626972_0 . . . 123
5.1771 so_24626972_24627084_0 . . . 123
5.1772 so_19300142_19636998_0 . . . 123
5.1773 so_19300142_19636998_1 . . . 123
5.1774 so_37134319_37134453_3 . . . 123
5.1775 so_28810470_28817538_2 . . . 123
5.1776 so_14730896_14732445_0 . . . 123
5.1777 so_35515702_35515916_0 . . . 123
5.1778 so_6594892_6595333_0 . . . 123
5.1779 so_27025553_27025662_0 . . . 123
5.1780 so_19569800_19570223_4 . . . 123
5.1781 so_37118426_37118638_1 . . . 123
5.1782 so_22371122_22372936_1 . . . 123
5.1783 so_2613387_2613393_1 . . . 123
5.1784 so_20275322_20735355_2 . . . 123
5.1785 so_46735324_46736040_1 . . . 123
5.1786 so_8309246_8309322_0 . . . 123
5.1787 so_15578935_15579050_2 . . . 123
5.1788 so_15578935_15579050_3 . . . 123
5.1789 so_15578935_15579185_0 . . . 123
5.1790 so_17409162_17409509_2 . . . 123
5.1791 so_14941723_14941807_2 . . . 123
5.1792 so_14941723_14941807_7 . . . 123
5.1793 so_14941723_14941807_1 . . . 123
5.1794 so_24471039_24471606_0 . . . 123
5.1795 so_28847860_28848091_0 . . . 123
5.1796 so_12260656_12260932_0 . . . 123
5.1797 so_11494403_11494486_1 . . . 123
5.1798 so_9018818_9018959_0 . . . 123
5.1799 so_22504496_22504862_0 . . . 123
5.1800 so_23829696_23831292_0 . . . 123
5.1801 so_32589328_32589328_1 . . . 123
5.1802 so_13995748_13996021_0 . . . 123
5.1803 so_16499683_16499784_0 . . . 123
5.1804 so_25638770_25639844_1 . . . 123
5.1805 so_33123520_33127426_0 . . . 123
5.1806 so_31882308_31882308_0 . . . 123
5.1807 so_39720871_39721564_2 . . . 123
5.1808 so_46021568_46027039_1 . . . 123
5.1809 so_25377877_25377970_0 . . . 123
5.1810 so_32134719_32134719_0 . . . 123
5.1811 so_27229578_27229664_0 . . . 123
5.1812 so_26917194_26917401_0 . . . 123
5.1813 so_24018770_24018870_1 . . . 123
5.1814 so_36040814_36043591_1 . . . 123
5.1815 so_23149343_23149518_1 . . . 123
5.1816 so_23149343_23149421_2 . . . 123
5.1817 so_19569800_19570223_5 . . . 123
5.1818 so_39508272_39508813_0 . . . 123
5.1819 so_45134220_45134428_1 . . . 123

5.1820 so_14125474_14125474_0 . . . 123
5.1821 so_14125474_14125911_1 . . . 123
5.1822 so_21535047_21536600_0 . . . 123
5.1823 so_19578237_19578336_0 . . . 123
5.1824 so_46612216_46612267_0 . . . 123
5.1825 so_15184050_15185323_0 . . . 123
5.1826 so_29096929_29097125_0 . . . 123
5.1827 so_19578237_19578336_1 . . . 123
5.1828 so_20714286_20714611_0 . . . 123
5.1829 so_14910616_14910723_3 . . . 123
5.1830 so_14910616_14910723_2 . . . 123
5.1831 so_14910616_14910723_4 . . . 123
5.1832 so_35551197_35553018_1 . . . 123
5.1833 so_45273291_45283790_2 . . . 123
5.1834 so_37134319_37134453_0 . . . 123
5.1835 so_37134319_37134453_2 . . . 123
5.1836 so_37134319_37134453_1 . . . 123
5.1837 so_14543801_14543811_1 . . . 123
5.1838 so_32845481_32847815_1 . . . 123
5.1839 so_32845481_32847815_0 . . . 123
5.1840 so_32845481_32847815_2 . . . 123
5.1841 so_19569800_19570223_7 . . . 123
5.1842 so_4582346_4582487_1 . . . 123
5.1843 so_4582346_4582487_2 . . . 123
5.1844 so_4582346_4582487_0 . . . 123
5.1845 so_30923296_30924216_0 . . . 123
5.1846 so_18791527_18791660_3 . . . 123
5.1847 so_18791527_18791660_0 . . . 123
5.1848 so_18791527_18791660_5 . . . 123
5.1849 so_18791527_18791660_4 . . . 123
5.1850 so_18791527_18791615_0 . . . 123
5.1851 so_18791527_18791660_2 . . . 123
5.1852 so_18791527_18791660_1 . . . 123
5.1853 so_32809304_32809350_1 . . . 123
5.1854 so_36158598_36158740_2 . . . 123
5.1855 so_36158598_36158775_0 . . . 123
5.1856 so_36158598_36158740_0 . . . 123
5.1857 so_36158598_36158740_1 . . . 123
5.1858 so_8365143_8365381_1 . . . 123
5.1859 so_19476818_19477123_1 . . . 123
5.1860 so_15283523_15284276_0 . . . 123
5.1861 so_28810470_28817538_4 . . . 123
5.1862 so_6494591_6494964_0 . . . 123
5.1863 so_19569800_19570223_8 . . . 123
5.1864 so_37118426_37118638_0 . . . 123
5.1865 so_26793072_26793255_0 . . . 123
5.1866 so_26793072_26793404_0 . . . 123
5.1867 so_26793072_26793233_1 . . . 123
5.1868 so_17409162_17409509_1 . . . 123
5.1869 so_36051008_36101569_0 . . . 123
5.1870 so_5013829_5014483_0 . . . 123

5.1871 so_21178903_21178903_0 . . . 123
5.1872 so_32070395_32071868_0 . . . 124
5.1873 so_24433250_24433518_3 . . . 124
5.1874 so_34536203_34536531_0 . . . 124
5.1875 so_36670354_36670545_0 . . . 124
5.1876 so_23754929_23755109_0 . . . 124
5.1877 so_23754929_23755100_0 . . . 124
5.1878 so_38702943_38702943_0 . . . 124
5.1879 so_12981147_12981183_0 . . . 124
5.1880 so_46761431_46761456_1 . . . 124
5.1881 so_37051273_37051384_0 . . . 124
5.1882 so_16505934_16505946_2 . . . 124
5.1883 so_17290376_17290493_0 . . . 124
5.1884 so_29883327_29883410_1 . . . 124
5.1885 so_34596933_34599085_1 . . . 124
5.1886 so_4042626_4042626_0 . . . 124
5.1887 so_16256065_16256081_0 . . . 124
5.1888 so_19208560_19208595_0 . . . 124
5.1889 so_6987497_6987624_0 . . . 124
5.1890 so_8663290_8663397_3 . . . 124
5.1891 so_25406818_25406930_0 . . . 124
5.1892 so_25406818_25406977_0 . . . 124
5.1893 so_34536203_34536531_1 . . . 124
5.1894 so_31882308_31882751_4 . . . 125
5.1895 so_31882308_31882751_0 . . . 125
5.1896 so_11053982_11054068_3 . . . 125
5.1897 Unary expressions . . . 125
5.1898 so_30360119_30360119_1 . . . 125
5.1899 so_40903533_40903668_2 . . . 126
5.1900 so_4285895_20614158_4 . . . 126
5.1901 so_15283523_15284276_8 . . . 126
5.1902 so_26128980_26129015_3 . . . 126
5.1903 so_33768926_33768940_1 . . . 126
5.1904 so_13842682_13843053_5 . . . 126
5.1905 so_24478830_24480354_1 . . . 126
5.1906 so_28573215_28573215_6 . . . 126
5.1907 so_18145567_18146631_2 . . . 126
5.1908 so_41097868_41098916_3 . . . 126
5.1909 so_26329999_26330301_0 . . . 126
5.1910 so_31362252_31362290_1 . . . 126
5.1911 so_21053273_21053756_0 . . . 126
5.1912 so_21053273_21053273_4 . . . 126
5.1913 so_35365205_35365205_0 . . . 126
5.1914 so_44046625_44046780_0 . . . 126
5.1915 so_32906247_32906350_1 . . . 126
5.1916 so_32906247_32906349_1 . . . 126
5.1917 so_25312225_25312375_6 . . . 126
5.1918 so_38615436_38615522_0 . . . 126
5.1919 so_13744760_13744760_0 . . . 126
5.1920 so_1110111_1110540_0 . . . 126
5.1921 so_30093923_30094209_2 . . . 126

5.1922 so_34882107_34883084_0 . . . 126
5.1923 so_38033305_38033403_0 . . . 126
5.1924 so_20242868_20243257_3 . . . 126
5.1925 so_20242868_20242868_1 . . . 126
5.1926 so_18965622_18966199_0 . . . 126
5.1927 so_988158_991310_2 . . . 126
5.1928 so_988158_991310_1 . . . 126
5.1929 so_42570752_42570876_1 . . . 126
5.1930 so_9048379_9048390_3 . . . 126
5.1931 so_16713990_16714015_0 . . . 126
5.1932 so_37977197_37977647_0 . . . 126
5.1933 so_36457841_36457841_2 . . . 126
5.1934 so_3028632_8621112_1 . . . 126
5.1935 so_8610571_8610728_12 . . . 126
5.1936 so_41996441_41996580_1 . . . 126
5.1937 so_16428234_16428253_1 . . . 126
5.1938 so_19033448_19037004_0 . . . 126
5.1939 so_21178903_21179084_0 . . . 126
5.1940 so_39860732_39861534_3 . . . 126
5.1941 so_39860732_39860802_1 . . . 126
5.1942 so_6085531_19257003_1 . . . 126
5.1943 so_44106654_44106781_0 . . . 126
5.1944 so_33161003_33161103_1 . . . 126
5.1945 so_35834275_35834337_1 . . . 126
5.1946 so_26329999_26330301_1 . . . 126
5.1947 so_24518517_24519164_7 . . . 126
5.1948 so_13803720_13803890_1 . . . 126
5.1949 so_25540033_25540631_5 . . . 126
5.1950 so_24411638_24412775_1 . . . 126
5.1951 so_3674456_40762790_0 . . . 126
5.1952 so_47201359_47201660_0 . . . 126
5.1953 so_2488515_2488515_0 . . . 126
5.1954 so_3868827_3868850_1 . . . 126
5.1955 so_18867889_18869108_1 . . . 126
5.1956 so_9963003_9963092_1 . . . 126
5.1957 so_26329999_26330301_2 . . . 126
5.1958 so_8380049_8380094_0 . . . 126
5.1959 so_22788026_22788099_0 . . . 126
5.1960 so_31518214_31518760_0 . . . 126
5.1961 so_21365490_21365587_1 . . . 126
5.1962 so_35365205_35365205_1 . . . 126
5.1963 so_38615436_38615522_1 . . . 126
5.1964 so_19340190_19340231_1 . . . 126
5.1965 so_14169752_14169973_1 . . . 126
5.1966 so_46750163_46750570_1 . . . 126
5.1967 so_46735324_46736040_4 . . . 126
5.1968 so_41131299_41131420_0 . . . 126
5.1969 so_34935618_34936548_0 . . . 126
5.1970 so_34512747_35002931_0 . . . 126
5.1971 so_988158_988210_0 . . . 126
5.1972 so_16715441_16715564_0 . . . 126

5.1973 so_5018603_5019603_0 . . . 126
5.1974 so_14400385_14543166_0 . . . 126
5.1975 so_5061275_5061303_0 . . . 126
5.1976 so_2280688_2280688_0 . . . 126
5.1977 so_40575060_40575405_1 . . . 126
5.1978 so_3810658_3810746_0 . . . 126
5.1979 so_41538954_41539043_1 . . . 126
5.1980 so_31754551_31754615_0 . . . 126
5.1981 so_11965402_11966326_0 . . . 126
5.1982 so_13803720_13803890_0 . . . 126
5.1983 so_14730534_14730687_1 . . . 126
5.1984 so_19564009_19565341_1 . . . 126
5.1985 so_19564009_19566059_1 . . . 126
5.1986 so_9114930_9115110_1 . . . 126
5.1987 so_38140630_38141463_0 . . . 126
5.1988 so_33073727_33073776_0 . . . 126
5.1989 so_25540033_25540631_6 . . . 126
5.1990 so_14290071_14290523_0 . . . 126
5.1991 so_46330784_46331111_1 . . . 126
5.1992 so_26346386_26346386_1 . . . 126
5.1993 so_20377210_20377285_4 . . . 126
5.1994 so_18312186_18312205_0 . . . 126
5.1995 so_7134197_7134218_0 . . . 126
5.1996 so_33344355_33344638_0 . . . 126
5.1997 so_8435654_8435826_0 . . . 126
5.1998 so_8435654_8435826_1 . . . 126
5.1999 so_43706852_43863930_0 . . . 126
5.2000 so_45816776_45817013_0 . . . 126
5.2001 so_34935618_34936548_1 . . . 126
5.2002 so_40819713_40819818_1 . . . 126
5.2003 so_14400385_14543166_1 . . . 126
5.2004 so_19541426_19541708_1 . . . 126
5.2005 so_16725111_16725135_0 . . . 126
5.2006 so_37653169_37653472_0 . . . 126
5.2007 so_7134261_7134261_1 . . . 126
5.2008 so_29181013_29181056_1 . . . 126
5.2009 so_20652424_20652665_1 . . . 126
5.2010 so_23038107_23038120_0 . . . 126
5.2011 so_30361112_30363234_0 . . . 126
5.2012 so_29712623_29712746_0 . . . 126
5.2013 so_41716138_41716138_1 . . . 126
5.2014 so_41716138_41716138_0 . . . 126
5.2015 so_17822131_17822241_0 . . . 126
5.2016 so_27275635_27275713_0 . . . 126
5.2017 so_27275635_27275780_0 . . . 126
5.2018 so_20442347_20444531_0 . . . 126
5.2019 so_25701381_25701418_0 . . . 126
5.2020 so_26495013_26495099_2 . . . 126
5.2021 so_24583476_24588157_1 . . . 126
5.2022 so_9474699_9474799_0 . . . 126
5.2023 so_9474699_9474699_0 . . . 126

5.2024 so_38028871_38028905_0 . . . 126
5.2025 so_8391732_8392047_0 . . . 126
5.2026 so_18720585_18720678_0 . . . 126
5.2027 so_26892674_26893482_0 . . . 126
5.2028 so_4267194_4267258_0 . . . 126
5.2029 so_14065337_14065392_0 . . . 126
5.2030 so_1269019_1269141_0 . . . 126
5.2031 so_1269019_1269049_4 . . . 126
5.2032 so_21242070_21242231_1 . . . 126
5.2033 so_45356673_45357246_0 . . . 126
5.2034 so_17313579_17313717_1 . . . 126
5.2035 so_32136092_32137530_5 . . . 127
5.2036 so_44679200_44679200_1 . . . 127
5.2037 so_35671630_35671651_0 . . . 127
5.2038 so_4966002_4966034_2 . . . 127
5.2039 so_16394959_16394987_0 . . . 127
5.2040 so_12993335_12993697_0 . . . 127
5.2041 so_30437431_30437435_8 . . . 127
5.2042 so_9833790_9833830_0 . . . 127
5.2043 so_46677148_46677408_0 . . . 127
5.2044 so_45676565_45676607_2 . . . 127
5.2045 so_4616755_4616903_1 . . . 127
5.2046 so_37055444_37056269_0 . . . 127
5.2047 so_30473958_30474166_0 . . . 127
5.2048 so_29402367_29402657_0 . . . 127
5.2049 so_26101257_26101357_0 . . . 127
5.2050 so_14778269_14778292_0 . . . 127
5.2051 so_32271852_32273373_0 . . . 127
5.2052 so_12730321_12730780_4 . . . 127
5.2053 so_8208023_8209275_0 . . . 127
5.2054 so_28141871_28142480_4 . . . 127
5.2055 so_35464793_35464872_0 . . . 127
5.2056 so_44880658_44880718_0 . . . 127
5.2057 so_19729591_19729681_1 . . . 127
5.2058 so_27835808_27838583_0 . . . 127
5.2059 so_3852768_3852774_4 . . . 127
5.2060 so_27375117_27375117_2 . . . 127
5.2061 so_27021465_27021971_4 . . . 127
5.2062 so_27021465_27021971_3 . . . 127
5.2063 so_44977326_44977601_1 . . . 127
5.2064 so_19482671_19482858_1 . . . 127
5.2065 so_19482671_19482688_0 . . . 127
5.2066 so_18258745_18258857_0 . . . 127
5.2067 so_21351799_21351799_2 . . . 127
5.2068 so_14798953_14799079_1 . . . 127
5.2069 so_34824323_34824404_1 . . . 127
5.2070 so_25654339_25656964_4 . . . 127
5.2071 so_35315472_35316231_0 . . . 127
5.2072 so_44698989_44699189_1 . . . 127
5.2073 so_8848364_8849453_0 . . . 127
5.2074 so_13651313_13651400_0 . . . 127

5.2075 so_3450420_3450554_1 . . . 127
5.2076 so_10509685_10511913_0 . . . 127
5.2077 so_20194009_20194172_0 . . . 127
5.2078 so_9560029_9589938_0 . . . 127
5.2079 so_21915630_21932717_0 . . . 127
5.2080 so_2461630_2461856_1 . . . 127
5.2081 so_6440812_6440812_1 . . . 127
5.2082 so_7721184_7721284_1 . . . 127
5.2083 so_12924694_12924707_0 . . . 127
5.2084 so_2944794_2945818_0 . . . 127
5.2085 so_26874140_26874374_2 . . . 127
5.2086 so_26228577_26228577_1 . . . 127
5.2087 so_28118360_28118454_1 . . . 127
5.2088 so_5976879_5976946_0 . . . 127
5.2089 so_32029213_32029296_1 . . . 127
5.2090 so_47444081_47444081_1 . . . 127
5.2091 so_12893954_12894647_0 . . . 127
5.2092 so_22878352_22878981_1 . . . 127
5.2093 so_33527566_33527633_2 . . . 127
5.2094 so_33527566_33527644_0 . . . 127
5.2095 so_19906802_19906869_2 . . . 127
5.2096 so_4562249_4562283_0 . . . 127
5.2097 so_44563748_44564204_1 . . . 127
5.2098 so_13120473_13122063_0 . . . 127
5.2099 so_43544550_43545676_3 . . . 127
5.2100 so_43544550_43545676_1 . . . 127
5.2101 so_29748189_29748285_0 . . . 127
5.2102 so_11774877_11775032_2 . . . 127
5.2103 so_39278200_39278200_0 . . . 127
5.2104 so_39278200_39390725_0 . . . 127
5.2105 so_39278200_39390725_1 . . . 127
5.2106 so_39278200_39278298_0 . . . 127
5.2107 so_39278200_39434983_0 . . . 127
5.2108 so_34540278_34540711_0 . . . 127
5.2109 so_34540278_34540325_0 . . . 127
5.2110 so_1666224_28259072_0 . . . 127
5.2111 so_12673735_12673749_0 . . . 127
5.2112 so_2632021_2632646_1 . . . 127
5.2113 so_46738487_46743940_2 . . . 127
5.2114 so_31079232_31079326_0 . . . 127
5.2115 so_37278537_37278537_2 . . . 127
5.2116 so_27380209_27380358_0 . . . 127
5.2117 so_18984934_18985085_0 . . . 127
5.2118 so_17673778_17673947_2 . . . 127
5.2119 so_17710421_17710673_1 . . . 127
5.2120 so_20552491_20552672_1 . . . 127
5.2121 so_28714018_28714908_0 . . . 127
5.2122 so_31860300_31860540_1 . . . 127
5.2123 so_12486285_12486300_0 . . . 127
5.2124 so_46457449_46458123_0 . . . 127
5.2125 so_37750904_37750904_2 . . . 127

5.2126 so_37750904_37750904_0 . . . 127
5.2127 so_4780605_4780865_1 . . . 127
5.2128 so_30206065_30206193_0 . . . 127
5.2129 so_44811869_44812577_0 . . . 127
5.2130 so_21364673_21364673_9 . . . 127
5.2131 so_29338126_29338126_1 . . . 127
5.2132 so_29338126_29339251_0 . . . 127
5.2133 so_22270918_22271436_12 . . . 127
5.2134 so_22270918_22271436_15 . . . 127
5.2135 so_22270918_22271436_13 . . . 127
5.2136 so_22270918_22271436_14 . . . 127
5.2137 so_876213_876216_1 . . . 127
5.2138 so_27840695_27840837_0 . . . 127
5.2139 so_42155680_42155724_0 . . . 127
5.2140 so_15934111_15943751_0 . . . 127
5.2141 so_46589959_46589959_0 . . . 127
5.2142 so_26200639_26201073_0 . . . 127
5.2143 so_28075100_28075212_0 . . . 127
5.2144 so_5067492_5067492_0 . . . 127
5.2145 so_10328554_10328628_0 . . . 127
5.2146 so_21411044_21411080_0 . . . 127
5.2147 so_4897844_4897859_0 . . . 127
5.2148 so_27851016_27851448_0 . . . 127
5.2149 so_47449598_47449750_1 . . . 127
5.2150 so_16484644_16485311_0 . . . 127
5.2151 so_18040393_32413257_1 . . . 127
5.2152 so_27464867_27464881_0 . . . 127
5.2153 so_27464867_27464885_0 . . . 127
5.2154 so_39211727_39211911_1 . . . 127
5.2155 so_33523585_33523811_0 . . . 127
5.2156 so_26403970_26404092_1 . . . 127
5.2157 so_26403970_26404028_0 . . . 127
5.2158 so_35088599_35088909_2 . . . 127
5.2159 so_34122434_34122776_0 . . . 127
5.2160 so_30746062_30746136_0 . . . 127
5.2161 so_26631169_26631169_0 . . . 127
5.2162 so_3931167_3931551_1 . . . 127
5.2163 so_27518251_27518372_1 . . . 127
5.2164 so_621616_621617_0 . . . 127
5.2165 so_4839626_4839659_0 . . . 127
5.2166 so_2944794_2945818_2 . . . 127
5.2167 so_26874140_26874374_1 . . . 127
5.2168 so_26228577_26228577_2 . . . 127
5.2169 so_46458363_46458404_0 . . . 127
5.2170 so_119123_37032302_0 . . . 127
5.2171 so_119123_37032302_5 . . . 127
5.2172 so_33647319_33648901_0 . . . 127
5.2173 so_35087542_35087542_0 . . . 127
5.2174 so_3225733_3225743_0 . . . 127
5.2175 so_13861161_13861244_0 . . . 127
5.2176 so_43544550_43545676_2 . . . 127

5.2177 so_19949737_19950151_0 . . . 127
5.2178 so_19949737_19950151_2 . . . 127
5.2179 so_38265936_38266419_0 . . . 127
5.2180 so_38265936_38266419_1 . . . 127
5.2181 so_11283361_11283393_1 . . . 127
5.2182 so_21844064_21844102_3 . . . 127
5.2183 so_2632021_2632646_0 . . . 127
5.2184 so_10446907_10446961_1 . . . 127
5.2185 so_10446907_10446961_0 . . . 127
5.2186 so_39165295_39165581_0 . . . 127
5.2187 so_39165295_39165501_0 . . . 127
5.2188 so_13077086_13077395_1 . . . 127
5.2189 so_3953154_3953226_0 . . . 127
5.2190 so_3953154_3953225_1 . . . 127
5.2191 so_39878237_40092051_0 . . . 127
5.2192 so_4780605_4780865_2 . . . 127
5.2193 so_55532_55660_0 . . . 127
5.2194 so_4351132_4351132_0 . . . 127
5.2195 so_39791806_39866710_2 . . . 127
5.2196 so_4637774_4638295_0 . . . 127
5.2197 so_4637774_4637870_0 . . . 127
5.2198 so_12979625_12980934_1 . . . 127
5.2199 so_30316966_30317069_1 . . . 127
5.2200 so_40476058_40476277_3 . . . 127
5.2201 so_21879691_21879799_0 . . . 127
5.2202 so_10348606_10348626_0 . . . 127
5.2203 so_32863272_32863623_0 . . . 127
5.2204 so_39319745_39319745_0 . . . 128
5.2205 so_27380209_27380358_1 . . . 128
5.2206 so_27851016_27851448_1 . . . 128
5.2207 so_41461175_41461201_1 . . . 128
5.2208 so_13980701_13980863_0 . . . 128
5.2209 so_27380209_27380358_2 . . . 128
5.2210 so_7788508_7788508_0 . . . 128
5.2211 so_7788508_7788551_0 . . . 128
5.2212 so_3628364_3628564_0 . . . 128
5.2213 so_24312885_24313226_0 . . . 128
5.2214 so_36991202_36991202_2 . . . 128
5.2215 so_20808905_20808905_0 . . . 128
5.2216 so_20808905_20809251_1 . . . 128
5.2217 so_36594569_36594669_0 . . . 128
5.2218 so_19870192_19870234_0 . . . 128
5.2219 so_30206065_30206193_1 . . . 128
5.2220 so_30206065_30206245_2 . . . 128
5.2221 so_33704978_33705667_1 . . . 128
5.2222 so_47122822_47122880_0 . . . 128
5.2223 so_20918981_20919283_0 . . . 128
5.2224 so_22798558_22798623_0 . . . 128
5.2225 so_22798558_22798623_1 . . . 128
5.2226 so_27901367_27901627_0 . . . 128
5.2227 so_5339950_5340164_0 . . . 128

5.2228 so_16775039_16775585_0 . . . 128
5.2229 so_35377671_35377775_0 . . . 128
5.2230 so_16749954_16750072_0 . . . 128
5.2231 so_10632142_10632142_0 . . . 128
5.2232 so_10632142_13081190_0 . . . 128
5.2233 so_19132671_19132953_1 . . . 128
5.2234 so_24248828_24248894_3 . . . 128
5.2235 so_26029277_26029456_4 . . . 128
5.2236 so_26957290_26957354_0 . . . 128
5.2237 so_15938077_15938077_0 . . . 128
5.2238 so_27595793_27595998_0 . . . 128
5.2239 so_33456141_33457935_4 . . . 128
5.2240 so_27556840_27557375_2 . . . 128
5.2241 so_9781723_9781886_4 . . . 128
5.2242 so_35389789_35389789_4 . . . 128
5.2243 so_17078002_17084365_3 . . . 128
5.2244 so_8720425_8735297_0 . . . 128
5.2245 so_15935080_15935163_1 . . . 128
5.2246 so_15935080_15935235_1 . . . 128
5.2247 so_26029277_26029456_6 . . . 129
5.2248 so_26566090_26566275_1 . . . 129
5.2249 so_26029277_26029456_11 . . . 129
5.2250 so_26029277_26029456_10 . . . 129
5.2251 so_26029277_26029456_8 . . . 129
5.2252 so_6972111_6972405_2 . . . 129
5.2253 so_26566090_26566275_6 . . . 129
5.2254 so_26566090_26566275_5 . . . 129
5.2255 so_26566090_26566275_3 . . . 129
5.2256 so_27901367_27902613_0 . . . 129
5.2257 so_33038802_33039043_0 . . . 129
5.2258 so_38601371_38601834_0 . . . 129
5.2259 so_12399628_12399812_0 . . . 129
5.2260 so_22125433_22126061_2 . . . 129
5.2261 so_22125433_22126061_3 . . . 129
5.2262 so_29637156_29638071_0 . . . 129
5.2263 so_29701416_29701482_0 . . . 129
5.2264 so_6177617_6177649_0 . . . 129
5.2265 so_4256817_4259436_1 . . . 129
5.2266 so_19467909_19467909_0 . . . 129
5.2267 so_15938077_15938077_2 . . . 129
5.2268 so_31550099_31550398_1 . . . 129
5.2269 so_26404384_26404545_0 . . . 129
5.2270 so_9781723_9781886_0 . . . 129
5.2271 so_17078002_17084365_0 . . . 129
5.2272 so_4643430_4643772_0 . . . 129
5.2273 so_24843541_24843581_0 . . . 129
5.2274 so_34209484_34209484_0 . . . 129
5.2275 so_33758838_33759010_0 . . . 129
5.2276 so_5559525_5559525_0 . . . 129
5.2277 so_26192284_26192303_5 . . . 129
5.2278 so_24151399_24151550_2 . . . 129

5.2279 so_31575140_31575140_0 . . . 129
5.2280 so_1087042_1087066_0 . . . 129
5.2281 so_4133422_4133555_0 . . . 129
5.2282 so_11074289_11074328_1 . . . 129
5.2283 so_11074289_11074289_0 . . . 129
5.2284 so_1136844_1136844_1 . . . 129
5.2285 so_17419639_17419953_0 . . . 129
5.2286 so_37574449_37574580_4 . . . 129
5.2287 so_37574449_37574580_2 . . . 129
5.2288 so_37516500_37516653_1 . . . 129
5.2289 so_37516500_37516653_2 . . . 129
5.2290 so_22453001_22453001_0 . . . 129
5.2291 so_33110423_33110548_2 . . . 129
5.2292 so_648380_648442_3 . . . 130
5.2293 so_17947193_17948221_0 . . . 130
5.2294 so_7058377_7058469_0 . . . 130
5.2295 so_15604411_15604573_4 . . . 130
5.2296 so_43915004_43915004_0 . . . 130
5.2297 so_43915004_43915421_0 . . . 130
5.2298 so_18354548_18354724_1 . . . 130
5.2299 so_13199551_13199801_1 . . . 130
5.2300 so_13199551_13199688_0 . . . 130
5.2301 so_37240340_37240483_3 . . . 130
5.2302 so_19657590_19657718_0 . . . 130
5.2303 so_30373_293047_0 . . . 130
5.2304 so_3802454_3806373_0 . . . 130
5.2305 so_17078002_17084365_1 . . . 130
5.2306 so_12536280_12536475_0 . . . 130
5.2307 so_6950055_6950099_0 . . . 130
5.2308 so_2393817_2394085_0 . . . 130
5.2309 so_13199551_13199801_2 . . . 130
5.2310 so_25875228_25875313_0 . . . 130
5.2311 so_3802454_3806373_1 . . . 130
5.2312 so_8220100_8220491_0 . . . 130
5.2313 so_30001927_30001987_0 . . . 130
5.2314 so_24981797_24981797_0 . . . 130
5.2315 so_47072261_47072594_2 . . . 130
5.2316 so_26895087_26895298_0 . . . 130
5.2317 so_47075871_47097868_0 . . . 130
5.2318 so_47075871_47076760_0 . . . 130
5.2319 so_34590885_34596711_2 . . . 130
5.2320 so_34590885_34596711_0 . . . 130
5.2321 so_31873616_31875487_0 . . . 130
5.2322 so_38680504_38680532_0 . . . 130
5.2323 so_1527680_4564433_0 . . . 131
5.2324 so_26614606_26615252_0 . . . 131
5.2325 so_16615128_16615144_0 . . . 131
5.2326 so_7054176_7054354_4 . . . 131
5.2327 so_18921089_18922076_2 . . . 131
5.2328 so_6495632_6495913_8 . . . 131
5.2329 so_3874615_3874745_0 . . . 131

5.2330 so_39668561_39668976_0 . . . 131
5.2331 so_18456572_18460544_0 . . . 131
5.2332 so_25002044_25002044_0 . . . 131
5.2333 so_25002044_25002044_2 . . . 131
5.2334 so_16582644_16582644_0 . . . 131
5.2335 so_32991327_32991327_0 . . . 131
5.2336 so_13466556_14557329_0 . . . 131
5.2337 so_45309711_45309794_0 . . . 131
5.2338 so_33486001_33486001_1 . . . 131
5.2339 so_19351082_19351104_0 . . . 131
5.2340 so_6973995_16510895_1 . . . 131
5.2341 so_5721532_5721788_0 . . . 131
5.2342 so_32991327_32991327_1 . . . 131
5.2343 so_6496912_6497126_0 . . . 131
5.2344 so_4684719_4684860_0 . . . 131
5.2345 so_24934981_24935360_0 . . . 131
5.2346 so_29707685_29708019_3 . . . 131
5.2347 so_29707685_29708019_5 . . . 131
5.2348 so_29707685_29708019_2 . . . 131
5.2349 so_29707685_29708019_4 . . . 131
5.2350 so_29707685_29708019_1 . . . 131
5.2351 so_3090268_3090308_0 . . . 131
5.2352 so_16518894_16519079_0 . . . 131
5.2353 so_15254_15948_0 . . . 131
5.2354 so_4684719_4684860_1 . . . 131
5.2355 so_14624215_14624536_0 . . . 131
5.2356 so_46696675_46697160_1 . . . 131
5.2357 so_46696675_46697160_2 . . . 131
5.2358 so_19351082_19351104_2 . . . 131
5.2359 so_19351082_19351104_1 . . . 131
5.2360 so_19351082_19351104_3 . . . 131
5.2361 so_17078002_17084365_2 . . . 131
5.2362 so_8720425_36854950_1 . . . 131
5.2363 so_8720425_8720425_2 . . . 131
5.2364 so_8720425_8720425_1 . . . 131
5.2365 so_8720425_8720425_0 . . . 131
5.2366 so_8720425_36854950_0 . . . 131
5.2367 so_5721532_5722017_2 . . . 131
5.2368 so_5721532_5722017_0 . . . 131
5.2369 so_33137115_33137151_2 . . . 131
5.2370 so_33137115_33137151_1 . . . 131
5.2371 so_33137115_33137151_0 . . . 131
5.2372 so_3811132_3811169_0 . . . 132
5.2373 so_11514185_11514528_0 . . . 132
5.2374 so_5889691_5890000_0 . . . 132
5.2375 so_26419786_26420073_1 . . . 132
5.2376 so_17571103_17573387_0 . . . 132
5.2377 so_17571103_17571103_0 . . . 132
5.2378 so_24853959_24854152_1 . . . 132
5.2379 so_11781724_11782277_2 . . . 132
5.2380 so_17589601_17590822_0 . . . 132

5.2381 so_17571103_17573387_2 . . . 132
5.2382 so_8456236_8458776_2 . . . 132
5.2383 so_2468367_2468538_0 . . . 132
5.2384 so_17104209_17104608_4 . . . 132
5.2385 so_17104209_17104608_3 . . . 132
5.2386 so_17104209_17104573_1 . . . 132
5.2387 so_17104209_17104608_2 . . . 132
5.2388 so_20452967_20453967_3 . . . 132
5.2389 so_5887416_5887445_0 . . . 132
5.2390 so_13593165_13593507_0 . . . 132
5.2391 so_8697105_8697182_0 . . . 132
5.2392 so_27258364_27329244_2 . . . 132
5.2393 so_29765961_29766171_0 . . . 132
5.2394 so_15604411_15604573_6 . . . 132
5.2395 so_19449330_19450808_0 . . . 132
5.2396 so_7546620_7546745_1 . . . 132
5.2397 so_14220573_14220679_0 . . . 132
5.2398 so_24248828_24248894_2 . . . 132
5.2399 so_11729603_11729729_4 . . . 132
5.2400 so_27698364_27698788_2 . . . 132
5.2401 so_27698364_27698788_1 . . . 132
5.2402 so_27698364_27698788_0 . . . 132
5.2403 so_25278066_25278066_0 . . . 132
5.2404 so_25278066_25278066_2 . . . 132
5.2405 so_25278066_25278066_1 . . . 132
5.2406 so_29291371_29292092_0 . . . 132
5.2407 so_33456141_33457935_3 . . . 132
5.2408 so_25395297_25395408_0 . . . 132
5.2409 so_9894406_9894479_5 . . . 132
5.2410 so_9894406_9894479_4 . . . 132
5.2411 so_9894406_9894479_3 . . . 132
5.2412 so_25394665_25394763_0 . . . 132
5.2413 so_2204176_2204380_0 . . . 132
5.2414 so_32402842_32403394_3 . . . 132
5.2415 so_32402842_32403394_2 . . . 132
5.2416 so_32402842_32403394_1 . . . 132
5.2417 so_17770935_17773924_2 . . . 132
5.2418 so_17770935_17773924_0 . . . 132
5.2419 so_17770935_17773924_1 . . . 132
5.2420 so_30012733_30013834_1 . . . 132
5.2421 so_9502810_9502847_0 . . . 132
5.2422 so_16339811_16340318_0 . . . 132
5.2423 so_35383453_35383576_9 . . . 132
5.2424 so_14659752_14659931_2 . . . 132
5.2425 so_14659752_14659931_9 . . . 132
5.2426 so_43915004_43915421_2 . . . 132
5.2427 so_43915004_43915421_3 . . . 132
5.2428 so_44773631_44773683_0 . . . 132
5.2429 so_41779374_41779932_0 . . . 132
5.2430 so_41779374_41779863_0 . . . 132
5.2431 so_45995708_45995708_0 . . . 132

5.2432 so_1978709_24146155_0 . . . 132
5.2433 so_40991687_40991687_0 . . . 132
5.2434 so_12805671_12805759_0 . . . 132
5.2435 so_17658902_17659133_0 . . . 132
5.2436 so_17947193_17948221_1 . . . 132
5.2437 so_17947193_17948221_2 . . . 132
5.2438 so_6647542_6647610_0 . . . 132
5.2439 so_43915004_43915415_0 . . . 132
5.2440 so_43915004_43915421_1 . . . 132
5.2441 so_1306094_1675301_0 . . . 132
5.2442 so_26979464_26980546_0 . . . 132
5.2443 so_1674980_1675682_0 . . . 132
5.2444 so_19686769_19688690_2 . . . 132
5.2445 so_39330251_39330368_1 . . . 132
5.2446 so_39330251_39330368_0 . . . 132
5.2447 so_39330251_39330368_2 . . . 132
5.2448 so_31152553_31152553_0 . . . 132
5.2449 so_6950055_6950099_1 . . . 132
5.2450 so_24959192_24959519_0 . . . 133
5.2451 so_19686769_19688690_4 . . . 133
5.2452 so_5367674_5367792_2 . . . 133
5.2453 so_5367674_5367826_1 . . . 133
5.2454 so_25019041_25019041_0 . . . 133
5.2455 so_19686769_19688690_3 . . . 133
5.2456 so_39563948_39564002_1 . . . 133
5.2457 so_6084137_6084177_3 . . . 133
5.2458 so_6084137_6084177_2 . . . 133
5.2459 so_6172232_6172305_1 . . . 133
5.2460 so_35525922_35526054_2 . . . 133
5.2461 so_27713666_27713666_3 . . . 133
5.2462 so_9909439_9909559_0 . . . 133
5.2463 so_16585713_16585840_0 . . . 133
5.2464 so_16585713_16585713_0 . . . 133
5.2465 so_15016634_15016634_1 . . . 133
5.2466 so_12761875_12761979_0 . . . 133
5.2467 so_12761875_12761979_1 . . . 133
5.2468 so_34371901_34372209_0 . . . 133
5.2469 so_25037314_25037314_1 . . . 133
5.2470 so_25037314_25037314_0 . . . 133
5.2471 so_1293326_1708737_1 . . . 133
5.2472 so_17519292_17519743_1 . . . 133
5.2473 so_17519292_17519743_0 . . . 133
5.2474 so_2709283_2709295_1 . . . 133
5.2475 so_32792320_32792341_0 . . . 133
5.2476 so_7460242_7460258_1 . . . 133
5.2477 so_941832_941959_1 . . . 133
5.2478 so_3788005_3788075_0 . . . 133
5.2479 so_17609609_17609683_2 . . . 133
5.2480 so_22071087_22071134_0 . . . 133
5.2481 so_22071087_22071134_1 . . . 133
5.2482 so_8599225_8599332_1 . . . 133

5.2483 so_18911172_18911333_0 . . . 133
5.2484 so_18911172_18911333_2 . . . 133
5.2485 so_73134_73420_0 . . . 133
5.2486 so_15629530_15629573_1 . . . 133
5.2487 so_6783993_16888463_2 . . . 133
5.2488 so_6783993_16888463_0 . . . 133
5.2489 so_25734220_25734228_1 . . . 133
5.2490 so_14444251_14444466_1 . . . 133
5.2491 so_35765449_35765500_1 . . . 133
5.2492 so_6084137_6084177_4 . . . 133
5.2493 so_6172232_6177338_1 . . . 133
5.2494 so_6172232_6172232_1 . . . 133
5.2495 so_6420329_6420445_0 . . . 133
5.2496 so_6420329_6420445_1 . . . 133
5.2497 so_16100069_16100415_0 . . . 133
5.2498 so_615355_10259714_0 . . . 133
5.2499 so_25329576_25329576_3 . . . 133
5.2500 so_25329576_25329576_1 . . . 133
5.2501 so_5148656_5148693_0 . . . 133
5.2502 so_45689690_45690345_1 . . . 133
5.2503 so_33118895_33119136_0 . . . 133
5.2504 so_26080582_26080603_0 . . . 133
5.2505 so_2502004_2502067_0 . . . 133
5.2506 so_33758838_33758838_0 . . . 133
5.2507 so_29475395_29475610_0 . . . 133
5.2508 so_19564455_19565322_0 . . . 133
5.2509 so_33119348_33119476_1 . . . 133
5.2510 so_41099704_41100376_0 . . . 133
5.2511 so_29222319_29237475_0 . . . 133
5.2512 so_4826768_4826985_0 . . . 133
5.2513 so_70880_70941_0 . . . 133
5.2514 so_27258364_27329244_5 . . . 133
5.2515 so_6495632_6495913_2 . . . 133
5.2516 so_42926763_42926875_1 . . . 133
5.2517 so_15016634_15016634_2 . . . 133
5.2518 so_3821261_3821288_1 . . . 133
5.2519 so_30572201_30572232_0 . . . 133
5.2520 so_12139108_12139221_0 . . . 133
5.2521 so_34371901_34372209_3 . . . 133
5.2522 so_39992588_39993006_0 . . . 133
5.2523 so_37171370_37171600_0 . . . 133
5.2524 so_32507684_32508177_0 . . . 133
5.2525 so_17519292_17519743_2 . . . 133
5.2526 so_45447125_45447209_0 . . . 133
5.2527 so_30764045_30764117_2 . . . 133
5.2528 so_20489358_20489408_1 . . . 133
5.2529 so_4255598_4255636_0 . . . 133
5.2530 so_4255598_4255636_1 . . . 133
5.2531 so_21547180_21547181_0 . . . 133
5.2532 so_29709491_29709769_1 . . . 133
5.2533 so_16334231_16334279_2 . . . 133

5.2534 so_2140319_2140413_2 . . . 133
5.2535 so_17609609_17609644_0 . . . 133
5.2536 so_13692112_13692176_1 . . . 133
5.2537 so_13692112_13692176_0 . . . 133
5.2538 so_36274522_36274543_0 . . . 133
5.2539 so_44104156_44104354_0 . . . 133
5.2540 so_4804258_4804301_0 . . . 133
5.2541 so_24501573_24501664_1 . . . 133
5.2542 so_6731331_6731331_2 . . . 133
5.2543 so_6731331_6731331_1 . . . 133
5.2544 so_6731331_6731331_0 . . . 133
5.2545 so_47354881_47354881_1 . . . 133
5.2546 so_788482_788497_0 . . . 133
5.2547 so_12617408_12617974_0 . . . 133
5.2548 so_27133056_27133124_0 . . . 133
5.2549 so_16527829_16527861_0 . . . 133
5.2550 so_16725787_16725884_0 . . . 133
5.2551 so_7370154_7370211_0 . . . 133
5.2552 so_32867939_32868057_0 . . . 133
5.2553 so_33353135_33353276_0 . . . 133
5.2554 so_73134_73420_1 . . . 133
5.2555 so_15454403_15454460_1 . . . 133
5.2556 so_7530476_7531156_0 . . . 133
5.2557 so_4190703_4190733_0 . . . 133
5.2558 so_21500469_21500476_0 . . . 133
5.2559 so_6783993_16888463_3 . . . 133
5.2560 so_34316928_34317029_0 . . . 133
5.2561 so_16489623_16489637_0 . . . 133
5.2562 so_41800410_41800482_0 . . . 134
5.2563 so_36760797_36760926_0 . . . 134
5.2564 so_3109134_3109160_0 . . . 134
5.2565 so_6172232_6172305_0 . . . 134
5.2566 so_6172232_6177338_2 . . . 134
5.2567 so_8769467_8769578_0 . . . 134
5.2568 so_32663008_32663025_0 . . . 134
5.2569 so_8864311_8864388_2 . . . 134
5.2570 so_6355438_6355438_3 . . . 134
5.2571 so_16100069_16100415_1 . . . 134
5.2572 so_10775247_10775259_0 . . . 134
5.2573 so_10775247_10775266_0 . . . 134
5.2574 so_19339938_19339992_0 . . . 134
5.2575 so_4294066_4294108_0 . . . 134
5.2576 so_4294066_4294076_0 . . . 134
5.2577 so_4178889_4178919_1 . . . 134
5.2578 so_25220229_25220259_1 . . . 134
5.2579 so_27768468_27769245_1 . . . 134
5.2580 so_2502004_2502067_1 . . . 134
5.2581 so_15281198_15281308_0 . . . 134
5.2582 so_29841845_29841983_0 . . . 134
5.2583 so_16575566_16576673_0 . . . 134
5.2584 so_9909439_9909559_1 . . . 134

5.2585 so_4612583_4612680_0 . . . 134
5.2586 so_21538879_21538942_0 . . . 134
5.2587 so_3049806_3049820_0 . . . 134
5.2588 so_2100644_2100685_0 . . . 134
5.2589 so_27258364_27329244_6 . . . 134
5.2590 so_6495632_6496004_0 . . . 134
5.2591 so_8752111_8752126_0 . . . 134
5.2592 so_16859737_16860058_0 . . . 134
5.2593 so_4531252_4531346_1 . . . 134
5.2594 so_34493967_34494205_0 . . . 134
5.2595 so_12320135_12320205_0 . . . 134
5.2596 so_24359194_24359735_0 . . . 134
5.2597 so_5715923_5715960_2 . . . 134
5.2598 so_36680781_36680972_0 . . . 134
5.2599 so_14129486_14129525_0 . . . 134
5.2600 so_31796349_31796446_0 . . . 134
5.2601 so_1293326_1708737_0 . . . 134
5.2602 so_21937739_21937889_0 . . . 134
5.2603 so_26705521_26705571_0 . . . 134
5.2604 so_15023016_15023257_0 . . . 134
5.2605 so_19482464_19482509_1 . . . 134
5.2606 so_41113491_41113721_2 . . . 134
5.2607 so_41113491_41113647_0 . . . 134
5.2608 so_17274236_17274384_0 . . . 134
5.2609 so_17274236_17274398_0 . . . 134
5.2610 so_7754532_7754656_0 . . . 134
5.2611 so_3809740_3809792_0 . . . 134
5.2612 so_45256524_45256525_1 . . . 134
5.2613 so_13019129_13019208_3 . . . 134
5.2614 so_6528461_6528590_0 . . . 134
5.2615 so_29453928_29453959_4 . . . 134
5.2616 so_29453928_29453959_1 . . . 134
5.2617 so_2709283_2709295_0 . . . 134
5.2618 so_4365157_4365157_0 . . . 134
5.2619 so_41709089_41709124_0 . . . 134
5.2620 so_24854236_24854287_0 . . . 134
5.2621 so_7460242_7460258_0 . . . 134
5.2622 so_14731334_14731410_0 . . . 134
5.2623 so_12340229_12340240_0 . . . 134
5.2624 so_31664064_31664530_0 . . . 134
5.2625 so_2140319_2140421_0 . . . 134
5.2626 so_2140319_2140413_0 . . . 134
5.2627 so_7547538_7547626_0 . . . 134
5.2628 so_7547538_7547608_0 . . . 134
5.2629 so_941832_941959_0 . . . 134
5.2630 so_11892591_11893154_0 . . . 134
5.2631 so_11892591_11892645_0 . . . 134
5.2632 so_11183832_11183858_0 . . . 134
5.2633 so_2597116_2597128_0 . . . 134
5.2634 so_12703142_12703164_0 . . . 134
5.2635 so_19174529_19174627_0 . . . 134

5.2636 so_39632563_39632632_1 . . . 134
5.2637 so_14910368_14910501_0 . . . 134
5.2638 so_17268183_17268264_0 . . . 134
5.2639 so_6171814_6171814_0 . . . 134
5.2640 so_46605184_46605439_1 . . . 134
5.2641 so_8918791_8919369_0 . . . 134
5.2642 so_2969383_2969506_0 . . . 134
5.2643 so_8599225_8599332_2 . . . 134
5.2644 so_5970523_5970551_0 . . . 134
5.2645 so_5970523_5970739_0 . . . 134
5.2646 so_32867939_32868057_1 . . . 134
5.2647 so_30082821_30083102_0 . . . 134
5.2648 so_2065938_2065961_0 . . . 134
5.2649 so_15629530_15629573_0 . . . 134
5.2650 so_7197677_7197677_1 . . . 134
5.2651 so_42416258_42418091_0 . . . 134
5.2652 so_25858939_25859013_0 . . . 134
5.2653 so_22998364_23002021_5 . . . 134
5.2654 so_18550304_18550460_0 . . . 134
5.2655 so_1271307_1271953_0 . . . 134
5.2656 so_4023794_4023846_0 . . . 134
5.2657 so_4023794_4023878_0 . . . 134
5.2658 so_39082100_39082100_0 . . . 134
5.2659 so_4325154_4325223_0 . . . 134
5.2660 so_33800783_33800866_0 . . . 134
5.2661 so_25805314_25805877_1 . . . 134
5.2662 so_24841083_24841123_0 . . . 134
5.2663 so_3956807_3956807_1 . . . 134
5.2664 so_29062695_29062887_0 . . . 134
5.2665 so_26608611_26608611_0 . . . 134
5.2666 so_29366376_29366540_0 . . . 134
5.2667 so_2517245_2517455_0 . . . 134
5.2668 so_1611771_1611795_0 . . . 134
5.2669 so_12323028_12323818_0 . . . 134
5.2670 so_12764952_12764952_0 . . . 134
5.2671 so_47353696_47354104_0 . . . 134
5.2672 so_41120443_41125563_0 . . . 134
5.2673 so_39503203_39503203_0 . . . 134
5.2674 so_39503203_39503316_0 . . . 134
5.2675 so_1767679_1767770_0 . . . 134
5.2676 so_17911539_17911572_0 . . . 134
5.2677 so_33011014_33011089_0 . . . 134
5.2678 so_26699669_26699818_0 . . . 134
5.2679 so_25329576_25329711_0 . . . 134
5.2680 so_6506836_6507139_1 . . . 134
5.2681 so_6506836_6510356_0 . . . 134
5.2682 so_44798762_44798850_0 . . . 134
5.2683 so_13771823_13771838_2 . . . 134
5.2684 so_47278993_47279278_2 . . . 134
5.2685 so_15604411_15604573_7 . . . 134
5.2686 so_21077140_21077195_1 . . . 134

5.2687 so_27380859_27380937_0 . . . 134
5.2688 so_30418814_30418814_1 . . . 134
5.2689 so_45654835_45655866_2 . . . 134
5.2690 so_45654835_45655866_0 . . . 134
5.2691 so_11153863_11153923_0 . . . 134
5.2692 so_17344727_17344873_0 . . . 134
5.2693 so_24047340_24048582_0 . . . 134
5.2694 so_47354881_47355199_0 . . . 134
5.2695 so_16725787_16725884_2 . . . 134
5.2696 so_6169125_6169172_0 . . . 134
5.2697 so_6169125_6169143_0 . . . 134
5.2698 so_31101854_31102112_0 . . . 134
5.2699 so_6783993_16888463_6 . . . 134
5.2700 so_35525922_35526054_3 . . . 134
5.2701 so_25329576_25329711_1 . . . 134
5.2702 so_6506836_6510356_1 . . . 134
5.2703 so_6506836_6507139_0 . . . 134
5.2704 so_45689690_45690345_5 . . . 134
5.2705 so_45689690_45690345_6 . . . 134
5.2706 so_45689690_45690345_3 . . . 134
5.2707 so_45689690_45691283_1 . . . 134
5.2708 so_45689690_45690345_4 . . . 134
5.2709 so_45689690_45691283_0 . . . 134
5.2710 so_3391335_3392172_2 . . . 134
5.2711 so_3391335_3392172_0 . . . 134
5.2712 so_47278993_47279232_0 . . . 134
5.2713 so_15604411_15604573_9 . . . 134
5.2714 so_3821852_3822007_0 . . . 134
5.2715 so_47075871_47076760_3 . . . 134
5.2716 so_47075871_47076760_4 . . . 134
5.2717 so_47075871_47076760_1 . . . 134
5.2718 so_47075871_47076760_2 . . . 134
5.2719 so_42032760_42033473_0 . . . 134
5.2720 so_21606073_21606073_4 . . . 134
5.2721 so_21606073_21606073_2 . . . 134
5.2722 so_21606073_21606073_3 . . . 134
5.2723 so_45654835_45655866_6 . . . 134
5.2724 so_45654835_45655866_4 . . . 134
5.2725 so_45654835_45655866_1 . . . 134
5.2726 so_45654835_45655250_0 . . . 134
5.2727 so_45654835_45655866_7 . . . 134
5.2728 so_45654835_45655866_3 . . . 134
5.2729 so_45654835_45655866_5 . . . 134
5.2730 so_43504053_43504790_0 . . . 134
5.2731 so_6731331_6731484_0 . . . 134
5.2732 so_47354881_47355199_1 . . . 134
5.2733 so_47354881_47354936_0 . . . 134
5.2734 so_13792981_13793032_0 . . . 134
5.2735 so_27133056_27133124_2 . . . 134
5.2736 so_27133056_27133124_1 . . . 134
5.2737 so_16725787_16725884_3 . . . 134

5.2738 so_6783993_16888463_7 . . . 134
5.2739 so_6783993_16888463_8 . . . 135
5.2740 so_24603894_24604101_0 . . . 135
5.2741 so_34164076_34164076_0 . . . 135
5.2742 so_24940403_24940403_0 . . . 135
5.2743 so_25019041_25019598_0 . . . 135
5.2744 so_24770119_24770416_1 . . . 135
5.2745 so_24770119_24770416_0 . . . 135
5.2746 so_4191680_4191969_1 . . . 135
5.2747 so_19579435_19579479_0 . . . 135
5.2748 so_46457449_46458123_1 . . . 135
5.2749 so_15283523_15284276_4 . . . 135
5.2750 so_42523833_42524024_1 . . . 135
5.2751 so_43819314_44692728_1 . . . 135
5.2752 so_18150625_18150932_0 . . . 135
5.2753 so_33210169_33210300_1 . . . 135
5.2754 so_35088599_35088909_3 . . . 135
5.2755 so_33210169_33210300_2 . . . 135
5.2756 so_34555793_34556095_4 . . . 135
5.2757 so_34555793_34556095_0 . . . 135
5.2758 so_34555793_34558600_12 . . . 135
5.2759 so_43819314_44692728_6 . . . 135
5.2760 so_43819314_44692728_5 . . . 135
5.2761 so_43819314_44692728_3 . . . 135
5.2762 so_33210169_33210300_4 . . . 135
5.2763 so_33210169_33210300_3 . . . 135
5.2764 so_18326665_18326665_0 . . . 135
5.2765 Explicit type conversion (cast notation) . . . 136
5.2766 so_31574561_31574668_0 . . . 136
5.2767 so_46162766_46163082_0 . . . 136
5.2768 so_26480413_26481317_1 . . . 136
5.2769 so_40801765_40802025_1 . . . 136
5.2770 so_40801765_40802025_0 . . . 136
5.2771 so_41003088_41003364_2 . . . 136
5.2772 so_15343276_15343438_0 . . . 136
5.2773 so_5061275_5061303_1 . . . 136
5.2774 so_24518517_24519164_12 . . . 136
5.2775 so_21973064_21973403_1 . . . 136
5.2776 so_20871139_20871205_1 . . . 136
5.2777 so_20871139_20871187_2 . . . 136
5.2778 so_20871139_20871187_1 . . . 136
5.2779 so_2963396_2963451_0 . . . 136
5.2780 so_1635897_1635966_0 . . . 136
5.2781 so_28235805_28236231_0 . . . 136
5.2782 so_36508995_36509337_1 . . . 136
5.2783 so_8288502_8288548_1 . . . 136
5.2784 so_3642626_3642778_0 . . . 136
5.2785 so_8464400_8585258_3 . . . 136
5.2786 so_29180947_29181049_4 . . . 136
5.2787 so_18036138_18037104_0 . . . 136
5.2788 so_22107838_22108026_6 . . . 136

5.2789 so_6414981_6415142_6 . . . 136
5.2790 so_24303735_24312655_4 . . . 136
5.2791 so_10770044_10770061_9 . . . 136
5.2792 so_10770044_10770061_8 . . . 136
5.2793 so_10770044_10770061_7 . . . 136
5.2794 so_10770044_10770061_6 . . . 136
5.2795 so_3574059_3574209_6 . . . 136
5.2796 so_13906959_13907057_1 . . . 136
5.2797 so_27135677_27135791_3 . . . 136
5.2798 so_8806791_8806931_1 . . . 136
5.2799 so_17925124_17925216_1 . . . 136
5.2800 so_17925124_17925216_0 . . . 136
5.2801 so_13667830_13668172_2 . . . 136
5.2802 so_26733828_26741609_1 . . . 136
5.2803 so_674982_675042_1 . . . 136
5.2804 so_25096321_25096461_1 . . . 136
5.2805 so_25096321_25096461_0 . . . 136
5.2806 so_36327494_36327779_0 . . . 136
5.2807 so_36327494_36327931_2 . . . 136
5.2808 so_36327494_36327931_1 . . . 136
5.2809 so_36327494_36327779_1 . . . 136
5.2810 so_30126487_30126487_2 . . . 136
5.2811 so_8248191_8248235_0 . . . 136
5.2812 so_8248191_8248235_3 . . . 136
5.2813 so_8248191_8248235_1 . . . 136
5.2814 so_8248191_8248235_2 . . . 136
5.2815 so_26982648_26982722_0 . . . 136
5.2816 so_1635897_1636040_1 . . . 136
5.2817 so_3642626_3642778_1 . . . 136
5.2818 so_3642626_3642778_2 . . . 136
5.2819 so_3642626_3642778_3 . . . 136
5.2820 so_3642626_3642778_4 . . . 136
5.2821 so_3642626_3642778_5 . . . 136
5.2822 so_39266211_39266310_0 . . . 136
5.2823 so_2488418_2488525_0 . . . 136
5.2824 so_2488418_2488525_1 . . . 136
5.2825 so_25896482_25896661_1 . . . 136
5.2826 so_25896482_25896661_0 . . . 136
5.2827 so_23519074_23520312_0 . . . 136
5.2828 so_15187650_15188114_9 . . . 136
5.2829 so_15187650_15188114_7 . . . 136
5.2830 so_10369606_10376574_1 . . . 136
5.2831 so_9234217_9234287_1 . . . 137
5.2832 Pointer-to-member operators . . . 137
5.2833 so_18911905_18912005_1 . . . 137
5.2834 so_46750163_46750570_3 . . . 137
5.2835 so_32260570_32261099_0 . . . 137
5.2836 so_38033305_38033403_1 . . . 137
5.2837 so_39472903_39473175_0 . . . 137
5.2838 so_4295117_4297948_0 . . . 137
5.2839 so_16059116_16067411_1 . . . 137

5.2840 so_37662100_37662548_1 . . . 137
5.2841 so_46187833_46188056_1 . . . 137
5.2842 so_9890218_9890859_1 . . . 137
5.2843 so_2675228_2675229_0 . . . 137
5.2844 so_26082959_26082959_0 . . . 137
5.2845 so_26082959_26083391_0 . . . 137
5.2846 so_11581903_11661602_2 . . . 138
5.2847 so_2696864_2699558_0 . . . 138
5.2848 so_42113821_42113947_0 . . . 138
5.2849 so_28136547_28136600_0 . . . 138
5.2850 so_33916113_33916186_0 . . . 138
5.2851 Multiplicative operators . . . 138
5.2852 so_25302480_25302649_0 . . . 138
5.2853 so_25529991_25530021_0 . . . 138
5.2854 so_31662792_31666090_2 . . . 138
5.2855 so_31500787_31500956_0 . . . 138
5.2856 so_24168460_24169125_2 . . . 138
5.2857 so_24168460_24169125_1 . . . 138
5.2858 so_6102948_17560755_1 . . . 138
5.2859 so_8425118_8425141_0 . . . 138
5.2860 so_31702505_31702644_1 . . . 138
5.2861 so_13305310_13305423_0 . . . 138
5.2862 so_25529991_25529991_0 . . . 138
5.2863 so_25968047_25968224_0 . . . 138
5.2864 so_25968047_25968224_1 . . . 138
5.2865 so_29994598_29994621_0 . . . 138
5.2866 so_12232650_12232669_0 . . . 138
5.2867 so_44116820_44116935_1 . . . 138
5.2868 so_44116820_44116971_0 . . . 138
5.2869 so_46563494_46576503_0 . . . 138
5.2870 so_42926763_42926899_0 . . . 138
5.2871 so_17106448_17107043_3 . . . 138
5.2872 so_24218367_24218394_1 . . . 138
5.2873 so_24218367_24218394_2 . . . 138
5.2874 so_7594508_7594577_0 . . . 138
5.2875 so_22275684_22275718_0 . . . 138
5.2876 so_17769172_17774047_0 . . . 138
5.2877 so_30459773_30459947_0 . . . 138
5.2878 so_25984642_25984647_1 . . . 138
5.2879 so_13630042_13630606_0 . . . 138
5.2880 so_40483266_40490924_0 . . . 138
5.2881 so_33916113_33916186_1 . . . 138
5.2882 Additive operators . . . 138
5.2883 so_19311239_19311738_1 . . . 138
5.2884 so_25302480_25302649_2 . . . 138
5.2885 so_32477249_32479159_0 . . . 138
5.2886 so_33202400_33203861_5 . . . 138
5.2887 so_9278224_9278267_0 . . . 138
5.2888 so_39856970_39863945_2 . . . 138
5.2889 so_30035547_30035924_0 . . . 138
5.2890 so_31212944_31213186_0 . . . 138

5.2891 so_30804659_30805815_1 . . . 138
5.2892 so_36024047_36024121_1 . . . 138
5.2893 so_3100365_3100936_0 . . . 138
5.2894 so_41662100_41662321_1 . . . 138
5.2895 so_27931157_27931379_0 . . . 138
5.2896 so_42192396_42192415_0 . . . 138
5.2897 so_20407747_20407956_1 . . . 138
5.2898 so_31480671_31480717_1 . . . 138
5.2899 so_31480671_31480717_0 . . . 138
5.2900 so_13750296_13750503_1 . . . 138
5.2901 so_27405418_27405419_3 . . . 138
5.2902 so_31719660_31719723_0 . . . 138
5.2903 so_20649734_20729634_0 . . . 138
5.2904 so_20649734_20652162_0 . . . 138
5.2905 so_26143976_26144237_0 . . . 138
5.2906 so_26143976_26144079_3 . . . 138
5.2907 so_4234120_4235537_3 . . . 138
5.2908 so_17665110_17665110_0 . . . 138
5.2909 so_25216568_25216568_0 . . . 138
5.2910 so_26590267_26590268_6 . . . 138
5.2911 so_31480671_31480717_2 . . . 139
5.2912 so_33202400_33203861_6 . . . 139
5.2913 so_5099769_21680645_0 . . . 139
5.2914 so_28779803_28779803_0 . . . 139
5.2915 so_24442725_24442725_0 . . . 139
5.2916 so_8404598_8404676_3 . . . 139
5.2917 so_26626663_26627168_2 . . . 139
5.2918 so_38441390_38441572_0 . . . 139
5.2919 so_9932242_9932371_7 . . . 139
5.2920 so_26128980_26129015_1 . . . 139
5.2921 so_41932588_41932707_0 . . . 139
5.2922 so_41932588_41932676_0 . . . 139
5.2923 so_19048667_19048781_0 . . . 139
5.2924 so_30089415_30089452_1 . . . 139
5.2925 so_37981617_37981656_0 . . . 139
5.2926 so_10539140_10539616_0 . . . 139
5.2927 so_26742432_26742541_1 . . . 139
5.2928 so_9114657_9114776_0 . . . 139
5.2929 so_9114657_9114715_1 . . . 139
5.2930 so_9114657_9114776_1 . . . 139
5.2931 so_2405555_2405597_0 . . . 139
5.2932 so_39401136_39401437_2 . . . 139
5.2933 so_16301088_16301421_0 . . . 139
5.2934 so_8128168_8128390_0 . . . 139
5.2935 so_21866832_21867071_0 . . . 139
5.2936 so_25654339_25656964_8 . . . 139
5.2937 so_26626663_26627045_0 . . . 139
5.2938 so_26626663_26627168_0 . . . 139
5.2939 so_28408649_28408967_0 . . . 139
5.2940 so_41189577_41190253_0 . . . 139
5.2941 so_41189577_41190113_0 . . . 139

5.2942 so_40590216_40620526_2 . . . 139
5.2943 so_40590216_40590301_3 . . . 139
5.2944 so_38441390_38441572_1 . . . 139
5.2945 so_38305559_38306084_0 . . . 139
5.2946 so_15283523_15284276_7 . . . 139
5.2947 so_13116534_13116656_0 . . . 139
5.2948 so_46018409_46018409_1 . . . 139
5.2949 so_26128980_26129015_2 . . . 139
5.2950 so_41932588_41932707_1 . . . 139
5.2951 so_32193096_32193297_0 . . . 139
5.2952 so_18061394_18061598_1 . . . 139
5.2953 so_20873984_20874063_0 . . . 139
5.2954 so_30089415_30089452_2 . . . 139
5.2955 so_28573215_28573215_8 . . . 139
5.2956 so_2036104_2036104_0 . . . 139
5.2957 so_37981617_37981656_1 . . . 139
5.2958 so_22237123_22237155_1 . . . 139
5.2959 so_26742432_26742541_2 . . . 139
5.2960 so_30804659_30804659_0 . . . 139
5.2961 so_12976675_12977340_0 . . . 139
5.2962 so_13997130_13997196_0 . . . 139
5.2963 so_13997130_13998100_0 . . . 139
5.2964 so_13997130_13997210_0 . . . 139
5.2965 so_3838855_39605164_2 . . . 139
5.2966 so_47498585_47498585_0 . . . 139
5.2967 so_32906247_32906350_2 . . . 139
5.2968 so_32906247_32906349_2 . . . 139
5.2969 so_41419164_41419216_0 . . . 139
5.2970 so_2728299_2728427_1 . . . 139
5.2971 so_2728299_2728427_0 . . . 139
5.2972 so_26238279_26238334_0 . . . 139
5.2973 so_18727022_18727216_1 . . . 139
5.2974 so_18020332_18020414_1 . . . 139
5.2975 so_7191650_7236710_1 . . . 139
5.2976 so_38251844_38251844_0 . . . 139
5.2977 so_38499130_38500181_0 . . . 139
5.2978 so_32069280_32069494_1 . . . 139
5.2979 so_24187390_24188484_1 . . . 139
5.2980 so_39878237_39878591_1 . . . 139
5.2981 so_46014164_46015752_0 . . . 139
5.2982 so_23635310_23635362_0 . . . 139
5.2983 so_9114657_9114715_0 . . . 139
5.2984 so_16233868_16233954_1 . . . 139
5.2985 so_28920949_28921140_1 . . . 139
5.2986 so_988158_988220_0 . . . 139
5.2987 so_988158_988408_2 . . . 139
5.2988 so_988158_991310_3 . . . 139
5.2989 so_988158_988210_1 . . . 139
5.2990 so_15608366_15608422_0 . . . 139
5.2991 so_29243116_29251966_0 . . . 139
5.2992 so_29243116_29244316_1 . . . 139

5.2993 so_38506726_38507513_0 . . . 139
5.2994 so_33574112_33574112_2 . . . 139
5.2995 so_31480671_31480717_4 . . . 139
5.2996 so_21505447_21505491_0 . . . 139
5.2997 so_27881092_27881102_0 . . . 139
5.2998 so_35354141_35354234_0 . . . 139
5.2999 so_15560763_15560783_0 . . . 139
5.3000 so_4377308_4377469_0 . . . 139
5.3001 so_20649734_20729634_12 . . . 139
5.3002 so_29825352_30343753_1 . . . 139
5.3003 so_29825352_29825553_1 . . . 139
5.3004 so_24501573_24502115_1 . . . 139
5.3005 so_13792648_13792845_2 . . . 139
5.3006 so_24859092_24859898_5 . . . 139
5.3007 so_32045888_32046531_0 . . . 139
5.3008 so_15217572_15221498_2 . . . 139
5.3009 so_12665045_12665065_0 . . . 139
5.3010 so_12665045_12665045_0 . . . 139
5.3011 so_18781840_18782612_2 . . . 139
5.3012 so_18781840_18782612_3 . . . 139
5.3013 so_47094166_47094343_1 . . . 139
5.3014 so_47094166_47094204_3 . . . 139
5.3015 so_46448673_46452913_1 . . . 139
5.3016 so_8404598_8404917_0 . . . 139
5.3017 so_4709531_4709541_1 . . . 139
5.3018 so_37292114_37292160_0 . . . 139
5.3019 so_16567587_16567705_0 . . . 139
5.3020 so_37033682_37033859_1 . . . 139
5.3021 so_33202549_33202656_0 . . . 139
5.3022 so_30095650_30096649_0 . . . 139
5.3023 so_16739622_16740522_0 . . . 139
5.3024 so_23000629_23000771_1 . . . 139
5.3025 so_39860732_39861534_2 . . . 139
5.3026 so_39860732_39861534_1 . . . 139
5.3027 so_16234626_16234816_0 . . . 139
5.3028 so_26420348_26421176_1 . . . 139
5.3029 so_25654339_25656964_9 . . . 139
5.3030 so_30369357_30369357_0 . . . 139
5.3031 so_14905499_14905508_0 . . . 139
5.3032 so_32924282_32924330_1 . . . 139
5.3033 so_38089644_38089779_0 . . . 139
5.3034 so_25552822_25553119_1 . . . 139
5.3035 so_12167220_12167301_1 . . . 139
5.3036 so_36030781_36030938_0 . . . 139
5.3037 so_6842880_6843474_4 . . . 139
5.3038 so_44767368_44767469_1 . . . 139
5.3039 so_44767368_44767506_1 . . . 139
5.3040 so_8943261_8943287_12 . . . 139
5.3041 so_7958642_7958989_0 . . . 139
5.3042 so_30804659_30805815_0 . . . 139
5.3043 so_3838855_3839447_0 . . . 139

5.3044 so_39878237_39878237_0 . . . 139
5.3045 so_40329101_40329102_5 . . . 139
5.3046 so_16233868_16233954_3 . . . 139
5.3047 so_45765452_45765660_3 . . . 139
5.3048 so_44409207_44416191_2 . . . 139
5.3049 so_11985399_11985451_1 . . . 139
5.3050 so_24501573_24502115_0 . . . 139
5.3051 so_40098080_40098140_6 . . . 139
5.3052 so_15217572_15221498_0 . . . 139
5.3053 so_15217572_15217572_0 . . . 139
5.3054 so_8128168_8128390_2 . . . 139
5.3055 so_15548813_15548873_8 . . . 139
5.3056 so_15548813_15548978_8 . . . 139
5.3057 so_5246969_5247009_3 . . . 139
5.3058 so_5246969_5247009_2 . . . 139
5.3059 so_3767757_3767792_0 . . . 139
5.3060 so_19121890_19122919_5 . . . 139
5.3061 so_22849688_22849688_2 . . . 139
5.3062 so_30628742_30631419_2 . . . 139
5.3063 so_21866832_21866885_0 . . . 139
5.3064 so_21866832_21867071_1 . . . 139
5.3065 so_29746465_29746566_1 . . . 139
5.3066 so_29746465_29746566_2 . . . 139
5.3067 so_29746465_29746566_0 . . . 139
5.3068 so_4860086_4860140_0 . . . 139
5.3069 so_36517178_36517265_1 . . . 139
5.3070 so_14905499_14905508_1 . . . 139
5.3071 so_37051199_37052522_0 . . . 140
5.3072 so_37049338_37049926_0 . . . 140
5.3073 so_38669010_38669662_0 . . . 140
5.3074 so_33461724_33461842_1 . . . 140
5.3075 so_25438432_25438480_0 . . . 140
5.3076 so_30628742_30631419_3 . . . 140
5.3077 so_22471909_22472039_1 . . . 140
5.3078 so_28811034_28811449_3 . . . 140
5.3079 so_28811034_28811449_10 . . . 140
5.3080 so_8128168_8128295_0 . . . 140
5.3081 so_8128168_8128390_4 . . . 140
5.3082 so_27681633_27681669_1 . . . 140
5.3083 so_36517178_36517265_0 . . . 140
5.3084 so_9149294_9149308_1 . . . 140
5.3085 Shift operators . . . 140
5.3086 so_24489929_24490603_0 . . . 140
5.3087 so_24489929_24489971_0 . . . 140
5.3088 so_4330282_4330321_1 . . . 140
5.3089 so_4330282_4330321_0 . . . 140
5.3090 so_7544411_7544444_0 . . . 140
5.3091 so_14423453_14423482_1 . . . 140
5.3092 so_24100576_24100676_0 . . . 140
5.3093 so_24018770_24018870_0 . . . 140
5.3094 so_10499104_10499371_0 . . . 140

5.3095 so_10499104_10499451_0 . . . 140
5.3096 so_16880187_16880293_3 . . . 140
5.3097 so_16880187_16880293_1 . . . 140
5.3098 so_32067643_32067643_0 . . . 140
5.3099 so_31217053_31217061_0 . . . 140
5.3100 so_32684369_32684432_2 . . . 140
5.3101 so_12991507_13038457_0 . . . 140
5.3102 so_32800866_32800889_0 . . . 140
5.3103 so_30473958_30474166_1 . . . 140
5.3104 so_11656794_11656916_0 . . . 140
5.3105 so_33058340_33058368_0 . . . 140
5.3106 so_27084470_27084605_0 . . . 140
5.3107 so_9429156_9429228_1 . . . 140
5.3108 so_38558083_38558103_0 . . . 140
5.3109 so_7886504_7886604_0 . . . 140
5.3110 so_18790923_18790983_0 . . . 140
5.3111 so_33251441_33251720_0 . . . 140
5.3112 so_24518517_24519164_8 . . . 140
5.3113 so_16797838_16797906_0 . . . 140
5.3114 so_19592895_19593012_0 . . . 140
5.3115 so_19592895_19593012_1 . . . 140
5.3116 so_13087816_13087876_0 . . . 140
5.3117 so_46494713_46499414_0 . . . 140
5.3118 so_10748506_10748558_0 . . . 140
5.3119 so_35506314_35506593_0 . . . 140
5.3120 so_16073327_16073430_0 . . . 140
5.3121 so_30558722_30558757_0 . . . 140
5.3122 so_30558722_30558757_1 . . . 140
5.3123 so_47439616_47439700_0 . . . 140
5.3124 so_36925291_36925596_2 . . . 140
5.3125 so_4234120_4235537_5 . . . 140
5.3126 so_12730321_12730455_0 . . . 140
5.3127 so_46097424_46097534_0 . . . 140
5.3128 so_3394259_3394302_0 . . . 140
5.3129 so_3394259_3394323_0 . . . 140
5.3130 so_19636539_19636588_0 . . . 140
5.3131 so_19636539_19636580_1 . . . 140
5.3132 so_19636539_19636580_0 . . . 140
5.3133 so_8713490_8713579_2 . . . 140
5.3134 so_3557720_3557752_1 . . . 140
5.3135 so_35805837_35805912_0 . . . 140
5.3136 so_28141871_28142480_1 . . . 140
5.3137 so_33110423_33110548_0 . . . 140
5.3138 so_24489929_24490603_1 . . . 140
5.3139 so_30407589_30407641_0 . . . 140
5.3140 so_2194310_2194359_1 . . . 140
5.3141 so_12125650_12125836_9 . . . 140
5.3142 so_29160220_29160287_1 . . . 140
5.3143 so_7544411_7544444_1 . . . 140
5.3144 so_29483123_29483123_1 . . . 140
5.3145 so_14396858_14399510_0 . . . 140

5.3146 so_10499104_10499371_1 . . . 140
5.3147 so_47556024_47556329_0 . . . 140
5.3148 so_15597198_15597362_0 . . . 140
5.3149 so_26192284_26192303_2 . . . 140
5.3150 so_26192284_26193183_1 . . . 140
5.3151 so_26192284_26192492_0 . . . 140
5.3152 so_26192284_26193183_0 . . . 140
5.3153 so_23250651_23250909_0 . . . 140
5.3154 so_23250651_23250909_1 . . . 140
5.3155 so_33058340_33058368_1 . . . 140
5.3156 so_33251441_33255763_6 . . . 140
5.3157 so_33251441_33255763_4 . . . 140
5.3158 so_34589775_34589920_0 . . . 140
5.3159 so_3784996_3784996_0 . . . 140
5.3160 so_3784996_3785121_0 . . . 140
5.3161 so_16797838_16797906_1 . . . 140
5.3162 so_19592895_19593367_0 . . . 140
5.3163 so_19592895_19593012_2 . . . 140
5.3164 so_4269838_4269882_0 . . . 140
5.3165 so_38737024_38738152_1 . . . 140
5.3166 so_38737024_38738152_4 . . . 140
5.3167 so_34519334_34519365_3 . . . 140
5.3168 so_12730321_12730780_1 . . . 140
5.3169 so_12695943_12696029_2 . . . 140
5.3170 so_22904670_22904772_0 . . . 140
5.3171 so_26331035_26331035_0 . . . 140
5.3172 so_26331035_26331035_1 . . . 140
5.3173 so_26331035_26331170_1 . . . 140
5.3174 so_26331035_26331170_3 . . . 140
5.3175 so_23590571_23590638_0 . . . 140
5.3176 so_26319592_26319665_0 . . . 140
5.3177 so_26319592_26319665_1 . . . 140
5.3178 so_13951720_13951845_1 . . . 140
5.3179 so_28141871_28142480_2 . . . 140
5.3180 so_33110423_33110548_1 . . . 140
5.3181 so_15729765_15730750_0 . . . 140
5.3182 so_13858655_13858691_1 . . . 140
5.3183 so_29160220_29160271_0 . . . 140
5.3184 so_7544411_7544444_2 . . . 140
5.3185 so_32647374_32647491_0 . . . 140
5.3186 so_26474272_26474344_0 . . . 140
5.3187 so_14396858_14399510_1 . . . 140
5.3188 so_10499104_10499371_2 . . . 140
5.3189 so_12987834_12988714_0 . . . 140
5.3190 so_26192284_26192492_1 . . . 140
5.3191 so_27954788_27954851_0 . . . 140
5.3192 so_25054786_25055817_0 . . . 140
5.3193 so_23861640_23861758_1 . . . 140
5.3194 so_18631574_18631651_1 . . . 140
5.3195 so_4234120_4235537_7 . . . 140
5.3196 so_12730321_12730780_2 . . . 140

5.3197 so_12730321_12730455_1 . . . 140
5.3198 so_23590571_23590638_1 . . . 140
5.3199 so_45429780_45429894_1 . . . 140
5.3200 so_45429780_45429850_0 . . . 140
5.3201 so_13951720_13951845_2 . . . 140
5.3202 so_10681375_10681494_0 . . . 140
5.3203 Relational operators . . . 140
5.3204 so_5369770_5369794_0 . . . 140
5.3205 so_14388855_14388886_2 . . . 140
5.3206 so_14388855_14388886_1 . . . 140
5.3207 so_14388855_14388887_0 . . . 140
5.3208 so_28091563_28091563_0 . . . 140
5.3209 so_2464664_2464713_0 . . . 140
5.3210 so_26143976_26144079_0 . . . 140
5.3211 so_34148992_34153294_6 . . . 140
5.3212 so_30108274_30108422_0 . . . 141
5.3213 so_26590267_26590268_8 . . . 141
5.3214 so_26289918_26290247_0 . . . 141
5.3215 so_22471909_22472039_4 . . . 141
5.3216 so_22471909_22472039_3 . . . 141
5.3217 so_32193096_32193172_2 . . . 141
5.3218 so_32193096_32193172_1 . . . 141
5.3219 so_32193096_32193172_0 . . . 141
5.3220 so_44907303_44907759_1 . . . 141
5.3221 so_47610789_47612863_0 . . . 141
5.3222 so_45023674_45023729_3 . . . 141
5.3223 so_45023674_45023729_2 . . . 141
5.3224 so_45023674_45023729_0 . . . 141
5.3225 so_45023674_45023729_1 . . . 141
5.3226 so_28811034_28811449_13 . . . 141
5.3227 so_28811034_28811449_12 . . . 141
5.3228 so_28811034_28811449_6 . . . 141
5.3229 so_28811034_28811449_5 . . . 141
5.3230 so_39878237_40018150_1 . . . 141
5.3231 so_39878237_40092051_1 . . . 141
5.3232 so_23635310_23635362_1 . . . 141
5.3233 so_30599331_30599762_0 . . . 141
5.3234 so_12038084_12038228_1 . . . 141
5.3235 so_29746465_29746566_3 . . . 141
5.3236 so_26590267_26590268_11 . . . 141
5.3237 so_26590267_26590268_10 . . . 141
5.3238 so_26590267_26590268_12 . . . 141
5.3239 so_26590267_26590268_9 . . . 141
5.3240 so_47610789_47612863_1 . . . 141
5.3241 so_45023674_45023729_4 . . . 141
5.3242 so_26590267_26590268_13 . . . 141
5.3243 so_31225264_31225334_0 . . . 141
5.3244 so_20423108_20491870_2 . . . 141
5.3245 so_34148992_34153294_7 . . . 141
5.3246 so_22609342_22609514_1 . . . 141
5.3247 Equality operators . . . 141

5.3248 so_20423108_20491870_1 . . . 141
5.3249 so_45634466_45634768_1 . . . 141
5.3250 so_22609342_22609528_0 . . . 141
5.3251 so_26590267_26590268_5 . . . 141
5.3252 so_12167220_12167352_0 . . . 141
5.3253 so_14176320_14176510_0 . . . 141
5.3254 so_4909766_4909801_0 . . . 141
5.3255 so_36507520_36507662_0 . . . 141
5.3256 so_23149343_23149518_4 . . . 141
5.3257 so_31922063_31922278_0 . . . 141
5.3258 so_26533740_26535126_5 . . . 141
5.3259 so_30694069_30694084_4 . . . 141
5.3260 so_5322227_5322329_1 . . . 141
5.3261 so_24215683_24215721_0 . . . 141
5.3262 so_19134124_19134258_0 . . . 141
5.3263 so_13734282_13734434_0 . . . 141
5.3264 so_9086372_9086675_0 . . . 141
5.3265 so_5246969_5247009_5 . . . 141
5.3266 so_28431654_28431982_0 . . . 141
5.3267 so_27754055_27754244_2 . . . 141
5.3268 so_27754055_27754244_1 . . . 141
5.3269 so_4665709_4665867_1 . . . 141
5.3270 so_19134124_19134422_0 . . . 141
5.3271 so_29455791_29455791_2 . . . 141
5.3272 so_36024047_36024121_2 . . . 142
5.3273 so_39108471_39109030_0 . . . 142
5.3274 Bitwise AND operator . . . 142
5.3275 so_32684369_32684432_1 . . . 142
5.3276 so_24518517_24519164_10 . . . 142
5.3277 so_12730321_12730780_6 . . . 142
5.3278 so_34147715_34147715_0 . . . 142
5.3279 Bitwise exclusive OR operator . . . 143
5.3280 so_12695943_12696029_3 . . . 143
5.3281 so_24381523_24381563_0 . . . 143
5.3282 Bitwise inclusive OR operator . . . 143
5.3283 so_3357614_3357684_1 . . . 143
5.3284 so_12730321_12730780_8 . . . 143
5.3285 so_29394518_29394518_0 . . . 143
5.3286 Logical AND operator . . . 143
5.3287 so_7543001_7543549_0 . . . 143
5.3288 so_26942958_26942974_0 . . . 143
5.3289 so_42314796_42314796_1 . . . 143
5.3290 so_5473107_25555976_0 . . . 143
5.3291 so_18685788_18685907_1 . . . 143
5.3292 so_35941429_35941517_0 . . . 143
5.3293 so_13169239_13169251_0 . . . 143
5.3294 so_31052979_31053002_1 . . . 143
5.3295 so_17348622_17348661_1 . . . 143
5.3296 so_35729956_35730320_1 . . . 143
5.3297 so_31947916_31947990_0 . . . 143
5.3298 so_35912322_35912619_2 . . . 143

5.3299 so_42613866_42613928_0 . . . 143
5.3300 so_628526_628533_0 . . . 143
5.3301 so_19065571_19065622_3 . . . 143
5.3302 so_33145951_33146001_0 . . . 143
5.3303 so_22243721_22243785_0 . . . 143
5.3304 so_13434672_13543853_0 . . . 143
5.3305 so_36414959_36416452_0 . . . 143
5.3306 so_15869902_15869917_1 . . . 143
5.3307 so_36578111_36578229_0 . . . 143
5.3308 so_25672971_25673057_0 . . . 143
5.3309 so_5683026_5683030_0 . . . 143
5.3310 so_37575219_37575537_1 . . . 143
5.3311 so_12829008_12829045_0 . . . 143
5.3312 so_26942958_26942974_1 . . . 143
5.3313 so_10782863_10782972_0 . . . 143
5.3314 so_35729956_35730320_2 . . . 143
5.3315 so_35978601_35978659_0 . . . 143
5.3316 so_31947916_31947990_1 . . . 143
5.3317 so_10778627_10779755_0 . . . 143
5.3318 so_35912322_35912619_5 . . . 143
5.3319 so_35912322_35912619_3 . . . 143
5.3320 so_15869902_15869917_2 . . . 143
5.3321 so_37575219_37575537_2 . . . 143
5.3322 Logical OR operator . . . 143
5.3323 so_42314796_42314796_0 . . . 143
5.3324 so_34037853_34037914_1 . . . 143
5.3325 so_35941429_35941517_1 . . . 143
5.3326 so_13169239_13169251_1 . . . 143
5.3327 so_31052979_31053002_2 . . . 143
5.3328 so_28667924_28701901_1 . . . 143
5.3329 so_45471489_45471734_1 . . . 143
5.3330 so_19690826_19690894_1 . . . 143
5.3331 so_35912322_35912619_4 . . . 143
5.3332 so_42613866_42614016_1 . . . 143
5.3333 so_628526_628533_1 . . . 143
5.3334 so_33145951_33146001_1 . . . 143
5.3335 so_5683026_5683030_1 . . . 143
5.3336 so_13027504_13027523_0 . . . 143
5.3337 Conditional operator . . . 143
5.3338 so_777261_36810382_2 . . . 143
5.3339 so_40172606_40172686_1 . . . 143
5.3340 so_31052979_31053002_3 . . . 143
5.3341 so_33417436_33417492_0 . . . 143
5.3342 so_47511185_47511434_1 . . . 143
5.3343 so_47511185_47511585_1 . . . 143
5.3344 so_17665519_17665570_0 . . . 143
5.3345 so_16808709_16809114_2 . . . 143
5.3346 so_45640431_45640555_0 . . . 143
5.3347 so_25725936_25726087_1 . . . 143
5.3348 so_25725936_25726087_0 . . . 143
5.3349 so_45815301_45815368_0 . . . 143

5.3350 so_28114301_28114301_1 . . . 143
5.3351 so_12337868_12337868_1 . . . 144
5.3352 so_20848132_20848282_0 . . . 144
5.3353 so_20848132_20848261_1 . . . 144
5.3354 so_7957696_7957814_0 . . . 144
5.3355 so_25044767_25044958_1 . . . 144
5.3356 so_25044767_25044767_0 . . . 144
5.3357 so_32217700_32217804_1 . . . 144
5.3358 so_8516565_8516599_1 . . . 144
5.3359 so_8516565_8516599_3 . . . 144
5.3360 so_8516565_8516599_2 . . . 144
5.3361 so_38933823_38936111_0 . . . 144
5.3362 so_14405837_14406177_0 . . . 144
5.3363 so_40917184_40923625_4 . . . 144
5.3364 so_31126559_31128008_3 . . . 144
5.3365 so_25661223_25661238_1 . . . 144
5.3366 so_34785505_34786621_0 . . . 144
5.3367 so_39109710_39140477_1 . . . 144
5.3368 so_42276302_42276668_2 . . . 144
5.3369 so_42276302_42276668_0 . . . 144
5.3370 so_3312031_3312507_0 . . . 144
5.3371 so_1445274_1445327_1 . . . 144
5.3372 so_1445274_1445327_0 . . . 144
5.3373 so_40167231_40167573_0 . . . 144
5.3374 so_40167231_40168133_5 . . . 144
5.3375 so_40167231_40168133_4 . . . 144
5.3376 so_40167231_40168133_7 . . . 144
5.3377 so_40167231_40168133_8 . . . 144
5.3378 so_14623266_14623433_1 . . . 144
5.3379 so_31400109_31409151_0 . . . 144
5.3380 so_31400109_31409151_1 . . . 144
5.3381 so_12840818_12840945_0 . . . 144
5.3382 so_35139114_35139114_0 . . . 144
5.3383 so_34477383_34492547_3 . . . 144
5.3384 so_8324338_8324457_0 . . . 144
5.3385 so_31577762_31578597_8 . . . 144
5.3386 so_31577762_31578597_10 . . . 144
5.3387 so_31577762_31578597_9 . . . 144
5.3388 so_31577762_31578597_6 . . . 144
5.3389 so_31577762_31578597_7 . . . 144
5.3390 so_31577762_31578597_3 . . . 144
5.3391 so_31577762_31578597_4 . . . 144
5.3392 so_31577762_31578597_5 . . . 144
5.3393 so_41826412_41826643_0 . . . 144
5.3394 so_5479620_5479673_0 . . . 144
5.3395 so_5479620_5479673_1 . . . 144
5.3396 so_24706480_24706610_1 . . . 144
5.3397 so_24706480_24706610_0 . . . 144
5.3398 so_42421458_42422428_1 . . . 144
5.3399 so_42421458_42422428_0 . . . 144
5.3400 so_32217700_32217804_2 . . . 144

5.3401 so_14328034_14328143_0 . . . 144
5.3402 so_38933823_38936111_2 . . . 144
5.3403 so_38933823_38936111_4 . . . 144
5.3404 so_29381171_29381331_1 . . . 144
5.3405 so_29381171_29381331_0 . . . 144
5.3406 so_16808709_16809114_1 . . . 144
5.3407 so_16808709_16809114_0 . . . 144
5.3408 so_46516285_46516285_0 . . . 144
5.3409 so_46516285_46516285_1 . . . 144
5.3410 so_14405837_14406177_1 . . . 144
5.3411 so_28659804_28660475_13 . . . 144
5.3412 so_28659804_28660475_6 . . . 144
5.3413 so_8535226_8535257_4 . . . 144
5.3414 so_8535226_8535257_5 . . . 144
5.3415 so_8535226_8535257_6 . . . 144
5.3416 so_8535226_8535257_2 . . . 144
5.3417 so_8535226_8535257_1 . . . 144
5.3418 so_33301745_33302211_8 . . . 144
5.3419 so_22098834_22098970_3 . . . 144
5.3420 so_26188281_26188341_0 . . . 144
5.3421 so_20443919_20444106_0 . . . 144
5.3422 so_42958730_42958730_1 . . . 144
5.3423 so_34785505_34786621_1 . . . 144
5.3424 so_40167231_40168133_9 . . . 144
5.3425 so_10359590_10359808_5 . . . 144
5.3426 so_5479620_5479673_2 . . . 144
5.3427 so_24706480_24706610_2 . . . 144
5.3428 so_12837631_12838694_2 . . . 144
5.3429 so_42421458_42422428_2 . . . 144
5.3430 so_32217700_32217700_0 . . . 144
5.3431 so_32217700_32217804_3 . . . 144
5.3432 so_11914691_11915906_0 . . . 144
5.3433 so_30489726_30490256_0 . . . 144
5.3434 so_3257453_3257519_0 . . . 144
5.3435 so_3895840_3896363_0 . . . 144
5.3436 so_16808709_16809114_4 . . . 144
5.3437 so_46516285_46516285_3 . . . 144
5.3438 so_3909720_3909720_0 . . . 144
5.3439 so_14405837_14406177_2 . . . 144
5.3440 so_8535226_8535257_8 . . . 144
5.3441 so_28114301_28114301_2 . . . 144
5.3442 so_17322517_17323105_0 . . . 144
5.3443 so_27633297_27633297_2 . . . 144
5.3444 so_34785505_34786621_2 . . . 144
5.3445 so_42276302_42276668_3 . . . 144
5.3446 so_40167231_40168133_10 . . . 144
5.3447 so_4691056_4691274_0 . . . 144
5.3448 so_5479620_5479673_3 . . . 144
5.3449 so_24706480_24706610_3 . . . 144
5.3450 so_42421458_42422442_0 . . . 144
5.3451 so_42421458_42422428_7 . . . 144

5.3452 so_42421458_42422428_3 . . . 144
5.3453 so_32217700_32217804_4 . . . 144
5.3454 so_16808709_16809114_5 . . . 144
5.3455 so_41637152_41637450_0 . . . 144
5.3456 so_8535226_8535257_10 . . . 144
5.3457 so_27989031_27989291_5 . . . 144
5.3458 so_20443919_20444106_1 . . . 145
5.3459 so_34785505_34786621_3 . . . 145
5.3460 so_26559758_30424903_6 . . . 145
5.3461 so_7851021_7851319_0 . . . 145
5.3462 so_40167231_40168133_11 . . . 145
5.3463 so_35707401_35707581_0 . . . 145
5.3464 so_20848132_20848261_0 . . . 145
5.3465 so_5291008_5291024_0 . . . 145
5.3466 so_24706480_24706610_4 . . . 145
5.3467 so_12837631_12838694_0 . . . 145
5.3468 so_12837631_12838694_1 . . . 145
5.3469 so_42421458_42422428_8 . . . 145
5.3470 so_32217700_32217804_5 . . . 145
5.3471 so_32217700_32217700_1 . . . 145
5.3472 so_27879704_27880106_0 . . . 145
5.3473 so_30489726_30489806_0 . . . 145
5.3474 so_20019791_20020084_0 . . . 145
5.3475 so_6102948_17560755_0 . . . 145
5.3476 so_16808709_16809114_7 . . . 145
5.3477 so_16808709_16809114_6 . . . 145
5.3478 so_24371868_24372323_1 . . . 145
5.3479 so_14405837_14406177_5 . . . 145
5.3480 so_14405837_14406177_4 . . . 145
5.3481 so_14405837_14406177_6 . . . 145
5.3482 so_18679093_18679093_0 . . . 145
5.3483 Assignment and compound assignment operators . . . 145
5.3484 so_44341215_44348568_0 . . . 145
5.3485 so_44880658_44880743_0 . . . 145
5.3486 so_19729591_19729681_4 . . . 145
5.3487 so_36099212_36099281_4 . . . 145
5.3488 so_27835808_27839451_1 . . . 145
5.3489 so_27835808_27836016_2 . . . 145
5.3490 so_26694842_26694842_0 . . . 145
5.3491 so_26694842_26694842_1 . . . 145
5.3492 so_30521292_30521417_0 . . . 145
5.3493 so_42314796_42314796_2 . . . 145
5.3494 so_3852768_3852774_3 . . . 145
5.3495 so_19244627_19244795_0 . . . 145
5.3496 so_47600716_47601116_0 . . . 145
5.3497 so_20414883_20415441_0 . . . 145
5.3498 so_34523923_34524091_1 . . . 145
5.3499 so_4176328_46171943_1 . . . 145
5.3500 so_33598938_33599142_1 . . . 145
5.3501 so_33598938_33600274_3 . . . 145
5.3502 so_14935722_14991772_1 . . . 145

5.3503 so_4265167_4265192_2 . . . 145
5.3504 so_35136597_35136597_1 . . . 145
5.3505 so_35136597_35136597_2 . . . 145
5.3506 so_3437110_3437197_0 . . . 145
5.3507 so_22076038_22076139_1 . . . 145
5.3508 so_5472115_46613205_0 . . . 145
5.3509 so_31823874_31831435_0 . . . 145
5.3510 so_31063331_31063331_2 . . . 145
5.3511 so_11989319_11989428_0 . . . 145
5.3512 so_19069681_19069981_1 . . . 145
5.3513 so_3607974_3608062_2 . . . 145
5.3514 so_10655290_10655346_0 . . . 145
5.3515 so_10655290_10655884_5 . . . 145
5.3516 so_6807393_6807398_0 . . . 145
5.3517 so_25817494_25817552_1 . . . 145
5.3518 so_10778627_10779755_1 . . . 145
5.3519 so_15343276_15343438_1 . . . 145
5.3520 so_17400137_17403467_3 . . . 145
5.3521 so_39559526_39559581_1 . . . 145
5.3522 so_24194076_24218798_0 . . . 145
5.3523 so_24194076_24194076_1 . . . 145
5.3524 so_24194076_24206844_0 . . . 145
5.3525 so_42997644_42999312_4 . . . 145
5.3526 so_42997644_42999312_3 . . . 145
5.3527 so_25323202_25323426_0 . . . 145
5.3528 so_19586893_19670459_1 . . . 145
5.3529 so_19586893_19670459_2 . . . 145
5.3530 so_19586893_19586893_0 . . . 145
5.3531 so_19586893_19682808_0 . . . 145
5.3532 so_45981850_45982500_0 . . . 145
5.3533 so_28482809_28483256_14 . . . 145
5.3534 so_42602382_42602463_3 . . . 145
5.3535 so_42602382_42602473_0 . . . 145
5.3536 so_29313902_29313902_0 . . . 145
5.3537 so_40699917_40699964_0 . . . 145
5.3538 so_16520297_16520442_0 . . . 145
5.3539 so_34148992_34151014_9 . . . 145
5.3540 so_34148992_34153294_5 . . . 145
5.3541 so_17225778_17227822_3 . . . 145
5.3542 so_10030098_10030193_0 . . . 145
5.3543 so_14991219_14991219_1 . . . 145
5.3544 so_22609067_22609240_0 . . . 145
5.3545 so_22609067_22609184_0 . . . 145
5.3546 so_32414448_32414467_0 . . . 145
5.3547 so_10653903_10653994_2 . . . 145
5.3548 so_10653903_10654019_1 . . . 145
5.3549 so_10653903_10653994_1 . . . 145
5.3550 so_25654339_25654561_3 . . . 145
5.3551 so_25654339_25654561_2 . . . 145
5.3552 so_25654339_25654561_1 . . . 145
5.3553 so_25654339_25656964_1 . . . 145

5.3554 so_44698989_44699189_0 . . . 145
5.3555 so_31067653_31067961_2 . . . 145
5.3556 so_37043401_37055970_5 . . . 145
5.3557 so_43852065_43852811_1 . . . 145
5.3558 so_33161003_33161401_3 . . . 145
5.3559 so_24796670_24796821_1 . . . 145
5.3560 so_20871139_20871205_0 . . . 145
5.3561 so_16317370_16317519_0 . . . 145
5.3562 so_13981993_13982675_1 . . . 145
5.3563 so_13981993_13982128_0 . . . 145
5.3564 so_28138376_28138436_2 . . . 145
5.3565 so_13982644_13982644_0 . . . 145
5.3566 so_27115527_27115650_2 . . . 145
5.3567 so_40699917_40699964_1 . . . 145
5.3568 so_20850536_20999389_8 . . . 145
5.3569 so_31547466_31547722_6 . . . 145
5.3570 so_13981993_13982675_2 . . . 145
5.3571 so_12396132_12396208_5 . . . 145
5.3572 so_28779803_28779876_2 . . . 145
5.3573 so_32684369_32684463_0 . . . 145
5.3574 so_13982644_13982644_1 . . . 145
5.3575 so_37043401_37055970_2 . . . 145
5.3576 so_3437110_3437197_1 . . . 145
5.3577 so_31547466_31547722_7 . . . 145
5.3578 so_25360835_25362435_3 . . . 145
5.3579 so_28138376_28138436_4 . . . 146
5.3580 so_27835808_27848818_2 . . . 146
5.3581 so_32722789_32723582_1 . . . 146
5.3582 so_24690195_24690326_0 . . . 146
5.3583 so_24194076_24206844_2 . . . 146
5.3584 so_24194076_24196478_2 . . . 146
5.3585 so_24194076_24218798_1 . . . 146
5.3586 so_22817517_22817633_1 . . . 146
5.3587 so_25323202_25323426_4 . . . 146
5.3588 so_25323202_25323699_1 . . . 146
5.3589 so_12690413_12690984_1 . . . 146
5.3590 so_11813692_22003520_1 . . . 146
5.3591 so_8423713_8423713_1 . . . 146
5.3592 so_8423713_8423713_0 . . . 146
5.3593 so_9585990_9588075_2 . . . 146
5.3594 so_37482588_37502709_0 . . . 146
5.3595 so_34441422_34441528_0 . . . 146
5.3596 so_16169161_16169257_0 . . . 146
5.3597 so_37254813_37255880_0 . . . 146
5.3598 so_32648805_32648892_0 . . . 146
5.3599 so_2990760_2990760_0 . . . 146
5.3600 so_42602382_42602479_0 . . . 146
5.3601 so_27220074_27220435_0 . . . 146
5.3602 Comma operator . . . 146
5.3603 so_12136156_12136194_2 . . . 146
5.3604 so_12136156_12136194_1 . . . 146

5.3605 so_31313626_31313671_1 . . . 146
5.3606 so_7784774_7784819_7 . . . 146
5.3607 so_7784774_7784819_0 . . . 146
5.3608 so_7784774_7784819_2 . . . 146
5.3609 so_7784774_7784819_5 . . . 146
5.3610 so_7784774_7784819_4 . . . 146
5.3611 so_20309662_20309680_0 . . . 146
5.3612 so_31052979_31053002_4 . . . 146
5.3613 so_19069681_19069981_4 . . . 146
5.3614 so_12288131_12288131_0 . . . 146
5.3615 so_6331588_22031971_1 . . . 146
5.3616 so_3953154_3953230_3 . . . 146
5.3617 so_16798422_16799424_3 . . . 146
5.3618 so_9563600_9563914_4 . . . 146
5.3619 so_38501587_38501596_9 . . . 146
5.3620 so_26458656_26462432_3 . . . 146
5.3621 so_26458656_26458656_0 . . . 146
5.3622 so_26458656_26462432_1 . . . 146
5.3623 so_27263729_27263916_0 . . . 146
5.3624 so_23564044_23564720_0 . . . 146
5.3625 so_23564044_23564199_0 . . . 146
5.3626 so_36352202_36352400_0 . . . 146
5.3627 so_24077253_24077606_4 . . . 146
5.3628 so_29868792_29873914_2 . . . 146
5.3629 so_8465425_8465435_3 . . . 146
5.3630 so_8465425_8465435_1 . . . 146
5.3631 so_45815301_45815382_0 . . . 146
5.3632 so_31115415_31115462_0 . . . 146
5.3633 so_5401062_5401062_1 . . . 146
5.3634 so_20765045_20765055_1 . . . 146
5.3635 so_20765045_20765055_2 . . . 146
5.3636 so_41321335_41321400_1 . . . 146
5.3637 so_12136156_12136194_3 . . . 147
5.3638 so_12136156_12136194_4 . . . 147
5.3639 so_24116817_24116818_2 . . . 147
5.3640 so_24116817_24116818_3 . . . 147
5.3641 so_44296002_44296079_5 . . . 147
5.3642 so_44296002_44296079_6 . . . 147
5.3643 Constant expressions . . . 147
5.3644 so_25890784_25891133_0 . . . 147
5.3645 so_36990014_36990071_1 . . . 147
5.3646 so_9152580_9152753_1 . . . 147
5.3647 so_14085956_14086253_0 . . . 147
5.3648 so_25595742_25598405_4 . . . 147
5.3649 so_8400791_8401267_0 . . . 147
5.3650 so_29346820_29365435_3 . . . 147
5.3651 so_29346820_29365435_4 . . . 147
5.3652 so_29346820_29365435_1 . . . 147
5.3653 so_31735037_31740377_0 . . . 147
5.3654 so_27167958_27168696_0 . . . 147
5.3655 so_27167958_27168696_2 . . . 147

5.3656 so_16575534_16576184_3 . . . 147
5.3657 so_22176777_22177425_2 . . . 147
5.3658 so_22176777_22177425_1 . . . 147
5.3659 so_27586009_27586105_0 . . . 147
5.3660 so_28614591_28614591_0 . . . 147
5.3661 so_34240136_34240330_1 . . . 147
5.3662 so_38398475_38401144_2 . . . 147
5.3663 so_31388208_31390380_11 . . . 147
5.3664 so_7424647_7424647_1 . . . 147
5.3665 so_20694579_20694693_0 . . . 147
5.3666 so_32920229_32926471_4 . . . 147
5.3667 so_32920229_32926471_2 . . . 147
5.3668 so_32920229_32926471_0 . . . 147
5.3669 so_32920229_32926471_3 . . . 147
5.3670 so_33301552_33302195_4 . . . 147
5.3671 so_33301552_33302195_5 . . . 147
5.3672 so_37866375_37866503_0 . . . 147
5.3673 so_33872039_34017482_3 . . . 147
5.3674 so_34381236_34381236_2 . . . 147
5.3675 so_34381236_34381236_1 . . . 147
5.3676 so_42997847_43007131_0 . . . 147
5.3677 so_42997847_43007131_1 . . . 147
5.3678 so_42997847_43007131_3 . . . 147
5.3679 so_29900218_29900558_0 . . . 147
5.3680 so_41580022_41580022_0 . . . 147
5.3681 so_43528101_43528436_4 . . . 147
5.3682 so_38962226_38962465_3 . . . 147
5.3683 so_38962226_38962465_1 . . . 147
5.3684 so_38962226_38962465_0 . . . 147
5.3685 so_38962226_38962465_2 . . . 147
5.3686 so_31537359_31537359_3 . . . 147
5.3687 so_31537359_31537359_5 . . . 147
5.3688 so_31537359_31537359_4 . . . 147
5.3689 so_31537359_31537359_2 . . . 147
5.3690 so_31537359_31537359_0 . . . 147
5.3691 so_34448254_34448254_1 . . . 147
5.3692 so_34448254_34448254_0 . . . 147
5.3693 so_47269936_47269937_0 . . . 147
5.3694 so_37303968_37304416_1 . . . 147
5.3695 so_44386415_44401707_0 . . . 147
5.3696 so_27713666_27713666_5 . . . 147
5.3697 so_40088477_40089058_1 . . . 147
5.3698 so_40088477_40088477_0 . . . 147
5.3699 so_9152580_9152753_2 . . . 147
5.3700 so_27913603_27913893_1 . . . 147
5.3701 so_27913603_27913893_2 . . . 147
5.3702 so_22204292_22204439_0 . . . 147
5.3703 so_25152301_25152301_1 . . . 147
5.3704 so_30527865_30528095_1 . . . 147
5.3705 so_31375381_31384320_3 . . . 147
5.3706 so_31375381_31384320_0 . . . 147

5.3707 so_28436148_28436735_0 . . . 147
5.3708 so_7625952_8683842_0 . . . 147
5.3709 so_1583791_1584079_3 . . . 147
5.3710 so_18145567_18146631_5 . . . 147
5.3711 so_18145567_18146631_6 . . . 147
5.3712 so_5605142_5612606_3 . . . 147
5.3713 so_24580714_24580747_2 . . . 147
5.3714 so_24580714_24580747_0 . . . 147
5.3715 so_24580714_24580747_1 . . . 147
5.3716 so_12530408_12530499_0 . . . 147
5.3717 so_27708629_29457433_0 . . . 147
5.3718 so_19876691_19877048_0 . . . 147
5.3719 so_31526278_31526278_8 . . . 147
5.3720 so_31526278_31526370_5 . . . 147
5.3721 so_31526278_31526370_9 . . . 147
5.3722 so_31526278_31526278_4 . . . 147
5.3723 so_31526278_31526370_11 . . . 147
5.3724 so_31526278_31526370_10 . . . 147
5.3725 so_31526278_31526278_9 . . . 147
5.3726 so_31526278_31527061_1 . . . 147
5.3727 so_31526278_31526370_7 . . . 147
5.3728 so_31526278_31526370_6 . . . 147
5.3729 so_31526278_31526278_7 . . . 147
5.3730 so_31526278_31526278_6 . . . 147
5.3731 so_34323489_34324971_8 . . . 147
5.3732 so_34323489_34362729_5 . . . 147
5.3733 so_34323489_34362729_4 . . . 147
5.3734 so_34323489_34324971_5 . . . 147
5.3735 so_5099769_5099769_2 . . . 147
5.3736 so_5099769_5099769_5 . . . 147
5.3737 so_27909109_27909109_0 . . . 147
5.3738 so_33649370_33651899_4 . . . 147
5.3739 so_31922063_31922278_1 . . . 147
5.3740 so_31922063_31922278_2 . . . 147
5.3741 so_46253990_46261649_2 . . . 147
5.3742 so_42070462_42070921_1 . . . 147
5.3743 so_34091623_34369191_5 . . . 147
5.3744 so_33159200_33159350_2 . . . 147
5.3745 so_43745569_43748514_1 . . . 147
5.3746 so_9258525_9259119_0 . . . 147
5.3747 so_9258525_9320911_0 . . . 147
5.3748 so_34477383_34492547_1 . . . 147
5.3749 so_24489060_24489060_0 . . . 147
5.3750 so_37165073_37166407_2 . . . 147
5.3751 so_37165073_37166407_3 . . . 147
5.3752 so_24212792_24212940_0 . . . 147
5.3753 so_26529542_26530069_1 . . . 147
5.3754 so_32623906_32624191_2 . . . 147
5.3755 so_32623906_32624191_3 . . . 147
5.3756 so_46119548_46120142_1 . . . 147
5.3757 so_21319413_21319413_1 . . . 147

5.3758 so_14915238_14920864_4 . . . 147
5.3759 so_24492190_24492190_3 . . . 147
5.3760 so_24492190_24492190_1 . . . 147
5.3761 so_43119643_43120923_2 . . . 147
5.3762 so_47511185_47511585_0 . . . 147
5.3763 so_42799268_42799405_1 . . . 147
5.3764 so_44840066_44841232_1 . . . 147
5.3765 so_44840066_44841232_4 . . . 147
5.3766 so_45593044_45593720_1 . . . 147
5.3767 so_45593044_45593724_1 . . . 147
5.3768 so_34555793_34557658_0 . . . 147
5.3769 so_34561884_34675596_2 . . . 147
5.3770 so_34561884_34675596_4 . . . 147
5.3771 so_34561884_34601508_0 . . . 147
5.3772 so_28662784_28662784_4 . . . 147
5.3773 so_25595742_25598405_1 . . . 147
5.3774 so_25595742_25598405_0 . . . 147
5.3775 so_23250651_23250909_3 . . . 147
5.3776 so_23250651_23250909_2 . . . 147
5.3777 so_32608458_32608695_4 . . . 147
5.3778 so_28264279_28264640_1 . . . 147
5.3779 so_36405541_36407197_0 . . . 147
5.3780 so_34280729_34280783_10 . . . 147
5.3781 so_44995196_44996066_3 . . . 147
5.3782 so_44995196_44996066_1 . . . 147
5.3783 so_24152418_24361568_0 . . . 147
5.3784 so_11725881_11734112_0 . . . 147
5.3785 so_41253121_41253658_1 . . . 147
5.3786 so_27303650_27303980_1 . . . 147
5.3787 so_45882871_45883965_0 . . . 147
5.3788 so_16004264_16004686_2 . . . 147
5.3789 so_43043397_43045636_0 . . . 147
5.3790 so_23930319_23930395_0 . . . 147
5.3791 so_26200827_26201529_1 . . . 147
5.3792 so_26200827_26201529_2 . . . 147
5.3793 so_38851469_38853356_0 . . . 147
5.3794 so_25660234_25660498_2 . . . 147
5.3795 so_25660234_25660498_1 . . . 147
5.3796 so_17480440_17480790_0 . . . 147
5.3797 so_39614225_39614455_0 . . . 147
5.3798 so_10524598_10524622_1 . . . 147
5.3799 so_34465047_34465086_1 . . . 147
5.3800 so_34465047_34465086_0 . . . 147
5.3801 so_24194076_24194076_0 . . . 147
5.3802 so_37326442_37326681_3 . . . 147
5.3803 so_28916895_28917356_1 . . . 147
5.3804 so_28916895_28917356_3 . . . 147
5.3805 so_21897015_21897696_1 . . . 147
5.3806 so_33936295_33937081_1 . . . 147
5.3807 so_4810664_7439261_7 . . . 147
5.3808 so_30604686_30604988_6 . . . 147

5.3809 so_27292667_27292851_0 . . . 147
5.3810 so_23428684_23428685_2 . . . 147
5.3811 so_27689942_27689942_0 . . . 147
5.3812 so_23564044_23564720_3 . . . 147
5.3813 so_33949332_33950387_3 . . . 147
5.3814 so_43665610_43688785_1 . . . 147
5.3815 so_42100976_42212056_2 . . . 147
5.3816 so_46724260_46725055_5 . . . 147
5.3817 so_46724260_46725055_3 . . . 147
5.3818 so_30089131_30089692_1 . . . 147
5.3819 so_31527913_31529497_4 . . . 147
5.3820 so_31527913_31529497_6 . . . 147
5.3821 so_31527913_31529497_2 . . . 147
5.3822 so_14730534_14731279_1 . . . 147
5.3823 so_34272899_34273406_0 . . . 147
5.3824 so_34272899_34273406_1 . . . 147
5.3825 so_31410933_31410986_1 . . . 147
5.3826 so_24077253_24077253_0 . . . 147
5.3827 so_37536573_37536869_0 . . . 147
5.3828 so_37536573_37536718_0 . . . 147
5.3829 so_34211688_34214734_11 . . . 147
5.3830 so_18352385_18353789_5 . . . 147
5.3831 so_18352385_18353789_4 . . . 147
5.3832 so_37287942_37288127_0 . . . 147
5.3833 so_37287942_37288127_1 . . . 147
5.3834 so_20862774_20862774_8 . . . 147
5.3835 so_39381076_39381076_1 . . . 147
5.3836 so_33095999_33100488_0 . . . 147
5.3837 so_20968026_20968544_2 . . . 147
5.3838 so_20968026_20968544_1 . . . 147
5.3839 so_46961749_46967394_3 . . . 147
5.3840 so_46961749_46967394_2 . . . 147
5.3841 so_46961749_46967394_5 . . . 147
5.3842 so_46961749_46967394_4 . . . 147
5.3843 so_30549819_30549858_0 . . . 147
5.3844 so_46372614_46373136_1 . . . 147
5.3845 so_4999241_4999832_0 . . . 147
5.3846 so_26582875_26584275_0 . . . 147
5.3847 so_23763748_34477904_5 . . . 147
5.3848 so_11522399_11800201_2 . . . 147
5.3849 so_42575122_42575409_0 . . . 147
5.3850 so_22098834_22098970_1 . . . 147
5.3851 so_18903113_18915868_1 . . . 147
5.3852 so_18903113_18915868_13 . . . 147
5.3853 so_18903113_18915868_8 . . . 147
5.3854 so_36393588_36393902_0 . . . 147
5.3855 so_7421170_7424395_2 . . . 147
5.3856 so_7421170_7424395_4 . . . 147
5.3857 so_7421170_7424395_6 . . . 147
5.3858 so_7421170_7424395_3 . . . 147
5.3859 so_7421170_7424395_5 . . . 147

5.3860 so_7421170_21319606_0 . . . 147
5.3861 so_7421170_7424395_1 . . . 147
5.3862 so_46510531_46510612_0 . . . 147
5.3863 so_24398102_24398102_0 . . . 147
5.3864 so_24398102_24400015_0 . . . 147
5.3865 so_37526366_37526522_0 . . . 147
5.3866 so_37526366_37526522_2 . . . 147
5.3867 so_33916113_33916186_6 . . . 147
5.3868 so_33916113_33916186_5 . . . 147
5.3869 so_10369606_10376574_0 . . . 147
5.3870 so_41720176_41720560_3 . . . 149
5.3871 so_37290298_37290467_1 . . . 149
5.3872 so_38960126_38961143_1 . . . 149
5.3873 so_643763_643927_1 . . . 149
5.3874 so_27586009_27586105_1 . . . 149
5.3875 so_33301552_33302605_10 . . . 149
5.3876 so_33301552_33302195_1 . . . 149
5.3877 so_33872039_34017482_1 . . . 149
5.3878 so_34381236_34381236_0 . . . 149
5.3879 so_34397359_34397474_0 . . . 149
5.3880 so_26024942_26025026_1 . . . 149
5.3881 so_32253075_32660221_2 . . . 149
5.3882 so_6742386_6745554_1 . . . 149
5.3883 so_2423352_2423412_0 . . . 149
5.3884 so_34018046_34018046_0 . . . 149
5.3885 so_41058178_41058695_1 . . . 149
5.3886 so_8111357_8111507_0 . . . 149
5.3887 so_33957274_33958291_4 . . . 149
5.3888 so_17559165_17559165_2 . . . 149
5.3889 so_1951741_1952685_0 . . . 149
5.3890 so_21273829_21273849_0 . . . 149
5.3891 so_10721130_10721407_2 . . . 149
5.3892 so_10721130_10721407_1 . . . 149
5.3893 so_33838101_33838558_5 . . . 149
5.3894 so_33838101_33838558_10 . . . 149
5.3895 so_34360286_34360286_0 . . . 149
5.3896 so_34360286_34594252_0 . . . 149
5.3897 so_34360286_34594252_1 . . . 149
5.3898 so_34360286_34594252_2 . . . 149
5.3899 so_36405541_36406210_0 . . . 149
5.3900 so_36405541_36406210_3 . . . 149
5.3901 so_13493353_13493394_0 . . . 149
5.3902 so_37166131_37166537_0 . . . 149
5.3903 so_37166131_37166556_4 . . . 149
5.3904 so_43043397_43045261_4 . . . 149
5.3905 so_43043397_43045261_2 . . . 149
5.3906 so_7651035_7651475_0 . . . 149
5.3907 so_24345399_24346350_1 . . . 149
5.3908 so_10925459_10933049_2 . . . 149
5.3909 so_17253623_17253784_1 . . . 149
5.3910 so_1327806_1328583_0 . . . 149

5.3911 so_33949332_33949332_0 . . . 149
5.3912 so_42100976_42212056_1 . . . 149
5.3913 so_31527913_31529497_1 . . . 149
5.3914 so_31527913_31527913_0 . . . 149
5.3915 so_31410933_31410986_4 . . . 149
5.3916 so_42491290_42491290_4 . . . 149
5.3917 so_24826139_24826298_4 . . . 149
5.3918 so_24826139_24826298_2 . . . 149
5.3919 so_38602032_38604097_0 . . . 149
5.3920 so_20968026_20968544_0 . . . 149
5.3921 so_26309910_26309910_1 . . . 149
5.3922 so_25395672_25395833_0 . . . 149
5.3923 so_28184888_28184997_1 . . . 149
5.3924 so_5924870_6179994_0 . . . 149
5.3925 so_17658446_17660391_0 . . . 149
5.3926 so_28614591_28614785_0 . . . 149
5.3927 so_28614591_28614776_1 . . . 149
5.3928 so_33301552_33302195_2 . . . 149
5.3929 so_46688093_46688778_0 . . . 149
5.3930 so_31537359_31537359_6 . . . 149
5.3931 so_31537359_31537359_7 . . . 149
5.3932 so_31537359_31537359_8 . . . 149
5.3933 so_34448254_34448254_3 . . . 149
5.3934 so_41420301_42777137_0 . . . 149
5.3935 so_31526278_31526278_0 . . . 149
5.3936 so_31526278_31526278_1 . . . 149
5.3937 so_31526278_31526278_2 . . . 149
5.3938 so_31922063_31922278_3 . . . 149
5.3939 so_34091623_34369191_1 . . . 149
5.3940 so_34091623_34369191_2 . . . 149
5.3941 so_34091623_34091623_0 . . . 149
5.3942 so_34091623_34369191_3 . . . 149
5.3943 so_24489060_24489060_2 . . . 149
5.3944 so_25595742_25598405_6 . . . 149
5.3945 so_38041884_38042018_0 . . . 149
5.3946 so_34465047_34465086_2 . . . 149
5.3947 so_46208655_46208860_1 . . . 149
5.3948 so_46208655_46208860_0 . . . 149
5.3949 so_46208655_46208860_2 . . . 149
5.3950 so_33704978_33705079_0 . . . 149
5.3951 so_31527913_31527913_1 . . . 149
5.3952 so_31527913_31527913_3 . . . 149
5.3953 so_31527913_31527913_2 . . . 149
5.3954 so_35544254_35544976_0 . . . 149
5.3955 so_35544254_35544976_2 . . . 149
5.3956 so_35544254_35544976_1 . . . 149
5.3957 so_31519935_31519935_3 . . . 149
5.3958 so_31519935_31519935_1 . . . 149
5.3959 so_31519935_31519935_2 . . . 149
5.3960 so_46961749_46967394_6 . . . 149
5.3961 so_43060011_43061368_0 . . . 149

5.3962 so_37504353_37504414_2 . . . 149
5.3963 so_37504353_37504414_0 . . . 149
5.3964 so_37504353_37504414_1 . . . 149
5.3965 so_10369606_10376574_2 . . . 149
5.3966 so_13571749_13784100_3 . . . 149
5.3967 so_31748296_31749381_1 . . . 149
5.3968 so_2902917_8957166_1 . . . 149
5.3969 so_27597163_27597304_1 . . . 150
5.3970 so_34360286_34577436_0 . . . 150

6 Statements **151**
6.1 so_35542228_35542686_2 . . . 151
6.2 so_15568959_15568997_2 . . . 151
6.3 Labeled statement . . . 151
6.4 so_12807115_12807141_0 . . . 151
6.5 so_7334952_7334968_1 . . . 151
6.6 so_29023601_29023816_2 . . . 151
6.7 so_22774681_22775256_0 . . . 151
6.8 so_13661328_13661328_2 . . . 151
6.9 Expression statement . . . 151
6.10 so_44393873_44393928_0 . . . 151
6.11 so_29868792_29873914_4 . . . 151
6.12 so_29868792_29873914_3 . . . 151
6.13 so_22233769_22235528_1 . . . 151
6.14 so_22233769_22235528_2 . . . 151
6.15 Compound statement or block . . . 151
6.16 so_35542228_35542686_3 . . . 151
6.17 so_24293311_24313401_3 . . . 151
6.18 so_18880611_18880713_0 . . . 151
6.19 so_5561869_5561993_0 . . . 151
6.20 so_15777133_15779613_10 . . . 151
6.21 so_15777133_15779613_9 . . . 151
6.22 so_19344955_19345096_4 . . . 151
6.23 Selection statements . . . 152
6.24 so_35542228_35542686_1 . . . 152
6.25 so_28288110_28288167_0 . . . 152
6.26 so_35411163_35411289_0 . . . 152
6.27 so_4427228_4427272_0 . . . 152
6.28 so_10521856_10522106_0 . . . 152
6.29 so_31659123_31659788_1 . . . 152
6.30 so_4075276_4075307_1 . . . 152
6.31 so_10365901_10365916_0 . . . 152
6.32 so_15568959_15568997_1 . . . 152
6.33 so_15568959_15568997_0 . . . 152
6.34 so_29936940_29937137_0 . . . 152
6.35 so_46880578_46880634_2 . . . 152
6.36 so_46880578_46880634_0 . . . 152
6.37 so_28742464_28742521_2 . . . 152
6.38 so_28742464_28742521_1 . . . 152
6.39 so_17514704_17514704_0 . . . 152
6.40 so_27430640_27430640_2 . . . 152

6.41 so_12351460_12351531_0 . . . 152
6.42 so_17514704_17514704_1 . . . 152
6.43 so_12655284_12655339_0 . . . 152
6.44 so_12351460_12351531_1 . . . 152
6.45 so_18225721_18225722_0 . . . 152
6.46 so_13643649_13644612_1 . . . 152
6.47 so_4075276_4075307_2 . . . 152
6.48 so_34497208_34500892_0 . . . 152
6.49 so_40390583_40390760_0 . . . 153
6.50 so_23638282_23638529_0 . . . 153
6.51 so_38535292_38535667_0 . . . 153
6.52 so_12419384_12419608_0 . . . 153
6.53 so_15876960_15878243_2 . . . 153
6.54 so_4286535_4286549_2 . . . 153
6.55 so_6807393_6807398_1 . . . 153
6.56 so_4966002_4966034_0 . . . 153
6.57 so_31206205_31206579_0 . . . 153
6.58 so_31206205_31206606_2 . . . 153
6.59 so_31206205_31206741_0 . . . 153
6.60 so_10365901_10365916_1 . . . 153
6.61 so_19935868_19936210_4 . . . 153
6.62 so_19935868_19936210_1 . . . 153
6.63 so_33079486_33079574_7 . . . 153
6.64 so_7119427_7119427_1 . . . 153
6.65 so_11478962_11478962_1 . . . 153
6.66 so_24940631_24941098_2 . . . 153
6.67 so_16994263_16994295_1 . . . 153
6.68 so_6807393_6807398_3 . . . 153
6.69 so_10521856_10522106_3 . . . 153
6.70 so_24574238_24574238_0 . . . 153
6.71 so_22200844_22200879_0 . . . 153
6.72 so_18042223_18042253_0 . . . 153
6.73 so_24314951_24315013_1 . . . 153
6.74 so_29867018_29867092_0 . . . 153
6.75 so_18195312_18195408_4 . . . 153
6.76 so_9432226_10751009_0 . . . 153
6.77 so_9432226_10751009_2 . . . 153
6.78 so_24314951_24315013_6 . . . 153
6.79 so_24314951_24315013_5 . . . 153
6.80 so_24314951_24315013_4 . . . 153
6.81 so_24314951_24315013_7 . . . 153
6.82 so_24314951_24315013_2 . . . 153
6.83 so_29867018_29867126_0 . . . 153
6.84 so_29867018_29867126_1 . . . 153
6.85 so_32253075_32660221_0 . . . 153
6.86 so_32253075_32660221_1 . . . 153
6.87 so_25143860_25144037_1 . . . 153
6.88 so_25143860_25143950_0 . . . 153
6.89 so_25143860_25143950_1 . . . 153
6.90 so_25143860_25144037_0 . . . 153
6.91 so_6257807_6259317_3 . . . 153

6.92 so_24018164_24018660_3 . . . 153
6.93 so_24018164_24018660_4 . . . 153
6.94 so_24018164_24018660_5 . . . 153
6.95 so_24018164_24018660_6 . . . 153
6.96 so_15572657_15572720_0 . . . 153
6.97 so_15572657_15572720_2 . . . 153
6.98 so_31143642_31143687_0 . . . 153
6.99 so_24079001_24079001_1 . . . 153
6.100 so_24079001_24079001_0 . . . 153
6.101 so_33256160_33256418_0 . . . 153
6.102 so_31206205_31206606_0 . . . 153
6.103 so_3823033_3823454_2 . . . 153
6.104 so_3823033_3823454_0 . . . 153
6.105 so_25047109_25048219_5 . . . 153
6.106 so_25047109_25048219_0 . . . 153
6.107 so_25047109_25048348_0 . . . 153
6.108 so_6327223_6327520_0 . . . 153
6.109 so_6327223_6327520_1 . . . 153
6.110 so_6327223_6327306_2 . . . 153
6.111 so_2094776_10740312_3 . . . 153
6.112 so_2094776_10740312_1 . . . 153
6.113 so_31441541_31441994_0 . . . 153
6.114 so_24314638_24314853_3 . . . 153
6.115 so_24314638_24314853_2 . . . 153
6.116 so_24314638_24314853_5 . . . 153
6.117 so_24314638_24314853_4 . . . 153
6.118 so_24188226_24188973_6 . . . 153
6.119 so_24188226_24188973_3 . . . 153
6.120 so_24188226_24188973_4 . . . 153
6.121 so_24188226_24188973_1 . . . 153
6.122 so_19345347_19345393_0 . . . 153
6.123 so_19345347_19345391_0 . . . 153
6.124 so_18195312_18195408_7 . . . 153
6.125 so_12254644_12254665_0 . . . 153
6.126 so_36921814_36921864_1 . . . 153
6.127 so_36921814_36921864_0 . . . 153
6.128 so_13561290_13561346_0 . . . 153
6.129 so_13561290_13561330_0 . . . 153
6.130 so_33256160_33256418_2 . . . 153
6.131 so_31206205_31206579_1 . . . 153
6.132 so_31206205_31207433_2 . . . 153
6.133 so_31206205_31206606_4 . . . 153
6.134 so_44165548_44165799_0 . . . 154
6.135 so_31206205_31206626_0 . . . 154
6.136 so_23371952_23372350_0 . . . 154
6.137 Iteration statements . . . 154
6.138 so_35542228_35542686_0 . . . 154
6.139 so_21413069_21413176_0 . . . 154
6.140 so_27430640_27430640_4 . . . 154
6.141 so_12351460_12351766_0 . . . 154
6.142 so_28575953_28576210_3 . . . 154

6.143 so_12351460_12351901_0 . . . 154
6.144 so_12351460_12351766_2 . . . 154
6.145 so_47546111_47546260_0 . . . 154
6.146 so_2654378_2655054_1 . . . 154
6.147 so_18129174_18129401_1 . . . 154
6.148 so_27430640_27430680_0 . . . 155
6.149 so_190748_190768_0 . . . 155
6.150 so_27430640_27430680_1 . . . 155
6.151 so_17514704_17514704_2 . . . 155
6.152 so_38655326_38656351_4 . . . 155
6.153 so_12351460_12351531_2 . . . 155
6.154 so_28575953_28576210_4 . . . 155
6.155 so_34497208_34500892_1 . . . 155
6.156 so_20188241_20189131_0 . . . 155
6.157 so_24676876_24677865_0 . . . 155
6.158 so_40814778_40814797_5 . . . 156
6.159 so_885908_887298_2 . . . 156
6.160 so_42546159_42546274_3 . . . 156
6.161 so_5258081_5258160_0 . . . 156
6.162 so_15675556_15675997_0 . . . 156
6.163 so_14047577_14047592_0 . . . 156
6.164 so_20188241_20189131_1 . . . 156
6.165 so_3601154_3601237_0 . . . 156
6.166 so_17994460_17994775_1 . . . 156
6.167 so_16240687_16240790_3 . . . 156
6.168 so_16240687_16240790_1 . . . 156
6.169 so_47020276_47020888_0 . . . 156
6.170 so_47020276_47020785_4 . . . 156
6.171 so_47020276_47020785_3 . . . 156
6.172 so_22225148_22225238_3 . . . 156
6.173 so_22225148_22225238_2 . . . 156
6.174 so_25101430_25101457_3 . . . 156
6.175 so_25101430_25101457_2 . . . 156
6.176 so_25101430_25101457_1 . . . 156
6.177 so_25232435_25232675_1 . . . 156
6.178 so_25232435_25232675_2 . . . 156
6.179 so_25232435_25232492_0 . . . 156
6.180 so_25232435_25232675_3 . . . 156
6.181 so_16407212_16407244_3 . . . 156
6.182 so_16407212_16407244_1 . . . 156
6.183 so_22878352_22878981_3 . . . 156
6.184 so_27933307_27935812_0 . . . 156
6.185 so_31310175_33065133_0 . . . 156
6.186 so_16226150_16226151_2 . . . 156
6.187 so_16226150_16226151_6 . . . 156
6.188 so_34933294_34933361_0 . . . 156
6.189 so_35809460_35847688_1 . . . 156
6.190 so_21689388_21689452_2 . . . 156
6.191 so_20758064_20761976_1 . . . 156
6.192 so_20758064_20761976_6 . . . 156
6.193 so_20758064_20761976_5 . . . 156

6.194 so_20758064_20761976_4 . . . 156
6.195 so_20758064_20761976_3 . . . 156
6.196 so_31581880_31585629_1 . . . 156
6.197 so_31581880_31585629_2 . . . 156
6.198 so_45493829_45493969_0 . . . 156
6.199 so_27140778_27140852_0 . . . 156
6.200 so_27140778_27140852_1 . . . 156
6.201 so_20580138_20726854_2 . . . 156
6.202 so_20580138_20726854_1 . . . 156
6.203 so_26995999_26998804_6 . . . 156
6.204 so_26995999_26998804_4 . . . 156
6.205 so_26995999_26998804_1 . . . 156
6.206 so_26995999_26998804_3 . . . 156
6.207 so_13037490_13037490_1 . . . 156
6.208 so_13037490_13039796_0 . . . 156
6.209 so_13037490_13037490_0 . . . 156
6.210 so_24059532_24059774_2 . . . 156
6.211 so_24059532_24059774_1 . . . 156
6.212 so_8164567_8165154_0 . . . 156
6.213 so_2648878_2648878_0 . . . 156
6.214 so_29990045_29990971_6 . . . 156
6.215 so_29990045_29990971_2 . . . 156
6.216 so_21139269_21139270_1 . . . 156
6.217 so_28242073_28242402_0 . . . 156
6.218 so_16607938_16607953_3 . . . 156
6.219 so_19052026_19052278_2 . . . 156
6.220 so_19052026_19052278_6 . . . 156
6.221 so_19052026_19052278_5 . . . 156
6.222 so_19052026_19052278_8 . . . 156
6.223 so_19052026_19052278_7 . . . 156
6.224 so_25814733_25814858_0 . . . 156
6.225 so_25814733_25814897_0 . . . 156
6.226 so_10821756_10822280_1 . . . 156
6.227 so_10821756_10822280_3 . . . 156
6.228 so_28143182_28143449_2 . . . 156
6.229 so_28143182_28143449_1 . . . 156
6.230 so_28143182_28143449_0 . . . 156
6.231 so_13548339_13548339_1 . . . 156
6.232 so_13548339_13548339_2 . . . 156
6.233 so_13057596_13057883_0 . . . 156
6.234 so_25232435_25232675_4 . . . 157
6.235 so_20758064_20761976_10 . . . 157
6.236 Jump statements . . . 157
6.237 so_17335671_17335758_0 . . . 157
6.238 so_7262818_7263017_0 . . . 157
6.239 so_8522055_8522271_0 . . . 157
6.240 so_8522055_8522271_1 . . . 157
6.241 so_3199995_3200128_1 . . . 157
6.242 so_15948381_15948444_1 . . . 157
6.243 so_3179936_3179974_0 . . . 157
6.244 so_5872484_5872710_0 . . . 157

6.245 so_13988811_13989085_1 . . . 157
6.246 so_11458830_11458901_0 . . . 157
6.247 so_19790804_19791281_11 . . . 157
6.248 so_7334952_7334968_2 . . . 157
6.249 so_7334952_7334968_4 . . . 157
6.250 so_43680035_43680348_0 . . . 157
6.251 so_14990584_14990634_1 . . . 157
6.252 so_32845362_32848577_0 . . . 157
6.253 so_32845362_32848577_1 . . . 157
6.254 so_9827376_9827438_0 . . . 157
6.255 so_3179494_3179559_0 . . . 157
6.256 so_12148965_12149008_0 . . . 157
6.257 so_16759733_16759922_0 . . . 157
6.258 so_26818106_26818124_0 . . . 157
6.259 so_6377767_6378064_0 . . . 157
6.260 so_2196327_2196495_7 . . . 157
6.261 so_6791215_6791537_1 . . . 157
6.262 so_6791215_6791435_1 . . . 157
6.263 so_8047743_8047778_2 . . . 157
6.264 so_18100976_18101094_1 . . . 157
6.265 so_21413069_21413160_0 . . . 157
6.266 so_19364635_19364732_0 . . . 157
6.267 so_19025352_19025518_4 . . . 157
6.268 so_13384587_13384601_0 . . . 157
6.269 so_16490478_16490491_0 . . . 157
6.270 so_16580510_16580557_0 . . . 157
6.271 so_32416410_32416426_2 . . . 158
6.272 so_9653722_9653800_0 . . . 158
6.273 so_12927835_12928089_0 . . . 158
6.274 so_12927835_12928089_1 . . . 158
6.275 so_20229010_20229637_0 . . . 158
6.276 so_38043588_38043643_0 . . . 158
6.277 so_21210626_21210704_0 . . . 158
6.278 so_1610030_1610148_5 . . . 158
6.279 so_1610030_1610454_0 . . . 158
6.280 so_30939147_30939437_0 . . . 158
6.281 so_2784075_2784137_0 . . . 158
6.282 so_2784075_2784093_0 . . . 158
6.283 so_9570823_9570905_1 . . . 158
6.284 so_18435962_18435978_1 . . . 158
6.285 so_25698481_25698523_0 . . . 158
6.286 so_2598084_2598093_0 . . . 158
6.287 so_20614282_20614325_0 . . . 158
6.288 so_30697941_30697998_1 . . . 158
6.289 so_15328244_15328382_1 . . . 158
6.290 so_13212901_13212916_1 . . . 158
6.291 so_25963952_25964025_0 . . . 158
6.292 so_22464103_22464892_1 . . . 158
6.293 so_18166650_18166673_0 . . . 158
6.294 so_21872972_21873375_1 . . . 158
6.295 so_27474578_27474747_0 . . . 158

6.296 so_17591253_17591362_0 . . . 158
6.297 so_36547048_36547150_4 . . . 158
6.298 so_293499_294183_0 . . . 158
6.299 so_11495292_11495319_0 . . . 158
6.300 so_8108117_8108117_0 . . . 158
6.301 so_20559603_20559713_0 . . . 158
6.302 so_15770613_15770796_0 . . . 158
6.303 so_21870290_21870350_0 . . . 158
6.304 so_17530279_17530324_0 . . . 158
6.305 so_17530279_17530324_1 . . . 158
6.306 so_8923018_8923762_2 . . . 158
6.307 so_43885606_43885724_0 . . . 158
6.308 so_27011854_27011882_0 . . . 158
6.309 so_27011854_27011854_0 . . . 158
6.310 so_35987493_35987535_2 . . . 158
6.311 so_35987493_35987890_3 . . . 158
6.312 so_35987493_35987896_2 . . . 158
6.313 so_35987493_35987535_0 . . . 158
6.314 so_16495794_16495812_0 . . . 158
6.315 so_16532712_16532764_0 . . . 158
6.316 so_3459810_3459887_0 . . . 158
6.317 so_21978188_21978207_0 . . . 158
6.318 so_23669716_23669716_2 . . . 158
6.319 so_25592317_25592331_1 . . . 158
6.320 so_34673460_34673980_0 . . . 158
6.321 so_34673460_34673460_1 . . . 158
6.322 so_3563001_3563124_0 . . . 158
6.323 so_37873303_37878471_0 . . . 158
6.324 so_3402178_3402194_0 . . . 158
6.325 so_3402178_3405364_0 . . . 158
6.326 so_9936011_9936077_1 . . . 158
6.327 so_9936011_9936084_0 . . . 158
6.328 so_9936011_9936090_1 . . . 158
6.329 so_12376461_16571948_1 . . . 158
6.330 so_12376461_12376488_1 . . . 158
6.331 so_34714541_34714878_1 . . . 158
6.332 so_41841057_41841115_0 . . . 158
6.333 so_20787171_20787187_0 . . . 158
6.334 so_15077466_15077788_2 . . . 158
6.335 so_8804612_8808865_3 . . . 158
6.336 so_8804612_8808865_4 . . . 158
6.337 so_24147653_24147696_1 . . . 158
6.338 so_15059250_15059275_0 . . . 158
6.339 so_16176072_16176088_1 . . . 158
6.340 so_15474512_15474559_1 . . . 158
6.341 so_35041577_35041788_0 . . . 158
6.342 so_35041577_35041790_0 . . . 158
6.343 so_22268667_22268729_1 . . . 158
6.344 so_16007233_16007270_1 . . . 158
6.345 so_16007233_16007270_0 . . . 158
6.346 so_15057718_15057734_0 . . . 158

6.347 so_25049280_25049289_0 . . . 158
6.348 so_24074392_24074411_0 . . . 158
6.349 so_32411066_32411666_0 . . . 158
6.350 so_15122764_15122912_0 . . . 158
6.351 so_39026390_39027129_1 . . . 158
6.352 so_19585507_42300052_0 . . . 158
6.353 so_32398088_32398177_0 . . . 158
6.354 so_46495209_46495303_0 . . . 158
6.355 so_15511541_15511594_0 . . . 158
6.356 so_23043042_23043061_0 . . . 158
6.357 so_4290473_4290993_0 . . . 158
6.358 so_30317541_30317741_3 . . . 158
6.359 so_9316717_9316717_1 . . . 158
6.360 so_9316717_9316717_0 . . . 158
6.361 so_11115922_11115968_1 . . . 158
6.362 so_6930419_6930463_0 . . . 158
6.363 so_6930419_6930483_0 . . . 158
6.364 so_24238786_24239020_1 . . . 158
6.365 so_27829608_27829747_0 . . . 158
6.366 so_27829608_27829747_2 . . . 158
6.367 so_19200076_19200230_0 . . . 158
6.368 so_13473154_13473277_0 . . . 158
6.369 so_24331640_24331698_1 . . . 158
6.370 so_40200024_40200243_0 . . . 158
6.371 so_25311671_25311761_0 . . . 158
6.372 so_27898343_27898401_1 . . . 158
6.373 so_27898343_27898401_0 . . . 158
6.374 so_11096027_11096091_0 . . . 158
6.375 so_9963974_9964002_0 . . . 158
6.376 so_9963974_9964276_0 . . . 158
6.377 so_42760220_42760220_0 . . . 158
6.378 so_26981433_26981594_0 . . . 158
6.379 so_37581041_37581235_0 . . . 158
6.380 so_32416410_32416426_3 . . . 158
6.381 so_32416410_32416426_1 . . . 158
6.382 so_11851468_11851757_0 . . . 158
6.383 so_27887026_27887447_0 . . . 158
6.384 so_25069405_25069585_0 . . . 158
6.385 so_16237650_16237714_0 . . . 158
6.386 so_19532286_19532322_0 . . . 158
6.387 so_31927128_31927151_1 . . . 158
6.388 so_25147667_25147859_0 . . . 158
6.389 so_41897130_41897477_0 . . . 158
6.390 so_16359089_16359154_0 . . . 158
6.391 so_26237432_26237876_0 . . . 158
6.392 so_18638286_18638372_0 . . . 158
6.393 so_5255777_5255839_3 . . . 158
6.394 so_5255777_5255839_1 . . . 158
6.395 so_2196327_2196495_5 . . . 158
6.396 so_5024218_5024585_0 . . . 158
6.397 so_32678334_32678381_0 . . . 158

6.398 so_10877955_10878440_1 . . . 158
6.399 so_30182097_30182400_0 . . . 158
6.400 so_23902485_23903299_0 . . . 158
6.401 so_5401062_5401062_2 . . . 158
6.402 so_17606306_17606373_0 . . . 158
6.403 so_39339768_39339883_1 . . . 158
6.404 so_39339768_39339840_0 . . . 158
6.405 so_3717497_3717844_0 . . . 158
6.406 so_15286450_15286450_8 . . . 158
6.407 so_15286450_15286450_0 . . . 158
6.408 so_28643552_28644192_0 . . . 158
6.409 so_35987493_35987925_0 . . . 158
6.410 so_35987493_35987910_3 . . . 158
6.411 so_20923800_20923847_0 . . . 158
6.412 so_27142891_27142942_0 . . . 158
6.413 so_3383090_3383099_0 . . . 158
6.414 so_17357199_17357266_0 . . . 158
6.415 so_34081317_34082287_0 . . . 158
6.416 so_43284908_43290109_0 . . . 158
6.417 so_5136295_5136332_0 . . . 158
6.418 so_22577880_22577909_0 . . . 158
6.419 Declaration statement . . . 158
6.420 so_34081317_34082287_1 . . . 158
6.421 so_8522055_8522198_1 . . . 159
6.422 so_15948381_15948444_0 . . . 159
6.423 so_24343724_24343764_0 . . . 159
6.424 so_34081317_34082287_2 . . . 159
6.425 so_11163212_11163410_2 . . . 159
6.426 so_15777133_15779613_8 . . . 159
6.427 so_16265507_16265514_0 . . . 159
6.428 so_20755236_20755302_1 . . . 159
6.429 so_9777561_9777679_0 . . . 159
6.430 so_14667813_14667830_1 . . . 159
6.431 so_19681203_19681261_0 . . . 159
6.432 so_2196327_2196495_9 . . . 159
6.433 so_6791215_6791537_2 . . . 159
6.434 so_8047743_8047778_1 . . . 159
6.435 so_3757445_3757620_1 . . . 159
6.436 so_3757445_3757620_0 . . . 159
6.437 so_8522055_8522198_2 . . . 159
6.438 so_8535731_8535804_1 . . . 159
6.439 so_22812183_22812183_2 . . . 159
6.440 so_44165548_44166698_1 . . . 159
6.441 so_44165548_44165799_1 . . . 159
6.442 so_34081317_34082287_4 . . . 159
6.443 so_34081317_34082287_3 . . . 159
6.444 so_4395289_4397136_1 . . . 159
6.445 so_4395289_4397136_0 . . . 159
6.446 so_7334952_7334968_3 . . . 159
6.447 so_28135802_28135866_0 . . . 159
6.448 so_39465121_39465466_1 . . . 159

6.449 so_39465121_39465558_0 . . . 159
6.450 so_22774681_22775256_1 . . . 159
6.451 so_22774681_22775256_2 . . . 159
6.452 so_29124145_29124251_1 . . . 159
6.453 so_5136295_5136332_1 . . . 159
6.454 so_38343358_38343407_0 . . . 159
6.455 so_6791215_6791435_0 . . . 159
6.456 so_6791215_6791537_0 . . . 159
6.457 so_92396_37368542_8 . . . 159
6.458 so_92396_37368542_7 . . . 159
6.459 so_246564_25796970_0 . . . 159
6.460 so_6009553_6009628_0 . . . 159
6.461 so_6009553_6009553_0 . . . 159
6.462 so_27286444_27286579_1 . . . 159
6.463 so_27286444_27286579_0 . . . 159
6.464 so_22902314_22904584_1 . . . 159
6.465 so_22902314_22904584_0 . . . 159
6.466 so_23842433_23842564_0 . . . 159
6.467 so_37075298_37075934_0 . . . 159
6.468 so_28453169_28453273_0 . . . 159
6.469 so_46700013_46700162_0 . . . 159
6.470 so_46700013_46700128_0 . . . 159
6.471 so_7567883_7569028_0 . . . 159
6.472 so_7567883_7569028_1 . . . 159
6.473 so_7567883_7569028_2 . . . 159
6.474 so_7567883_7569028_6 . . . 159
6.475 so_7567883_7569028_3 . . . 159
6.476 so_37177423_37179977_2 . . . 159
6.477 so_15894917_15895137_0 . . . 159
6.478 so_15894917_15895137_1 . . . 159
6.479 so_2939367_2939407_0 . . . 159
6.480 so_42080993_42081049_0 . . . 159
6.481 so_29230353_29230353_0 . . . 159
6.482 so_6982158_6982219_0 . . . 159
6.483 so_44165548_44166698_0 . . . 159
6.484 so_6189096_6189122_4 . . . 159
6.485 so_2955921_3082613_0 . . . 159
6.486 so_26055092_26055306_0 . . . 159
6.487 so_6086912_6099828_0 . . . 159
6.488 so_1661529_1661564_0 . . . 159
6.489 so_8697207_8697219_1 . . . 159
6.490 so_8697207_8697219_0 . . . 159
6.491 so_41724428_41725812_0 . . . 159
6.492 so_36421348_36422187_2 . . . 159
6.493 so_36421348_36422187_1 . . . 159
6.494 so_20826018_20826645_0 . . . 159
6.495 so_9057613_9057653_0 . . . 159
6.496 so_9057613_9057644_1 . . . 159
6.497 so_31041515_31041515_1 . . . 159
6.498 so_55510_58804_3 . . . 159
6.499 so_449436_449823_0 . . . 159

6.500 so_22783166_22783340_7 . . . 159
6.501 so_22783166_22783340_8 . . . 159
6.502 so_11711920_11711991_0 . . . 159
6.503 so_35889337_36109560_0 . . . 159
6.504 so_35889337_35991335_0 . . . 159
6.505 so_34256960_34256995_0 . . . 159
6.506 so_14607506_14607736_0 . . . 159
6.507 so_46790262_46790262_0 . . . 159
6.508 so_9795846_9796133_0 . . . 159
6.509 so_11666887_11667596_0 . . . 159
6.510 so_33397778_33398162_0 . . . 159
6.511 so_37650957_37651093_1 . . . 159
6.512 so_43609186_43609303_0 . . . 159
6.513 so_14244644_14244652_0 . . . 159
6.514 so_18140552_18140733_0 . . . 159
6.515 so_18140552_18140733_1 . . . 159
6.516 so_12302057_12302355_0 . . . 159
6.517 so_44698412_44698412_0 . . . 159
6.518 so_33689895_33690255_1 . . . 159
6.519 so_32872261_32872261_0 . . . 159
6.520 so_18304120_18304993_0 . . . 159
6.521 so_38860572_38860689_1 . . . 159
6.522 so_8391058_8391983_6 . . . 159
6.523 so_20755236_20755302_0 . . . 159
6.524 so_23856003_23856057_0 . . . 159
6.525 so_40821421_40821454_0 . . . 159
6.526 so_14861257_14861421_0 . . . 159
6.527 so_31443437_31443437_0 . . . 159
6.528 so_335369_335746_0 . . . 159
6.529 so_41486703_41489471_1 . . . 159
6.530 so_26759511_26759682_0 . . . 159
6.531 so_24173910_24174026_0 . . . 159
6.532 so_23593800_23594083_0 . . . 159
6.533 so_17407553_17408879_0 . . . 159
6.534 so_6620348_6626706_0 . . . 159
6.535 so_1338403_21907389_0 . . . 159
6.536 so_14172314_14172442_0 . . . 159
6.537 so_2666578_2806385_0 . . . 159
6.538 so_26816465_26818073_0 . . . 159
6.539 so_33033859_33033910_0 . . . 159
6.540 so_30885449_30885670_1 . . . 159
6.541 so_30885449_30885898_0 . . . 159
6.542 so_33114896_33115014_0 . . . 159
6.543 so_24531186_24531338_1 . . . 159
6.544 so_31018420_31018812_1 . . . 159
6.545 so_31018420_31018420_0 . . . 159
6.546 so_31018420_31018812_0 . . . 159
6.547 so_6967179_6967246_3 . . . 159
6.548 so_6967179_6967246_2 . . . 159
6.549 so_6967179_6967246_0 . . . 159
6.550 so_9058206_9058206_0 . . . 159

6.551 so_7789873_7809890_0 . . . 159
6.552 so_8395848_8395972_2 . . . 159
6.553 so_12546579_12547347_0 . . . 159
6.554 so_38130185_38132601_1 . . . 159
6.555 so_55510_58804_5 . . . 159
6.556 Ambiguity resolution . . . 160
6.557 so_25174676_25174744_1 . . . 160
6.558 so_34417071_34417330_0 . . . 160
6.559 so_5297763_5300674_0 . . . 160
6.560 so_28283215_28283405_0 . . . 160
6.561 so_17994955_18000366_0 . . . 160
6.562 so_41864822_41864990_1 . . . 160
6.563 so_30883473_30883739_0 . . . 160
6.564 so_6052857_6054646_1 . . . 160
6.565 so_45496423_45496586_0 . . . 160
6.566 so_30209962_30210182_0 . . . 160
6.567 so_24116817_24116818_4 . . . 160
6.568 so_47190568_47191759_0 . . . 160
6.569 so_29058769_29058856_0 . . . 160
6.570 so_24623071_24623317_6 . . . 160
6.571 so_27843375_27844266_1 . . . 160
6.572 so_18176117_18176272_1 . . . 160
6.573 so_16200933_16201230_0 . . . 160
6.574 so_31016784_31016833_0 . . . 160
6.575 so_24155571_24155937_1 . . . 160
6.576 so_28283215_28283405_7 . . . 160
6.577 so_17994955_18000366_1 . . . 160
6.578 so_30883473_30883739_3 . . . 160
6.579 so_44067289_44067863_1 . . . 160
6.580 so_44067289_44067863_0 . . . 160
6.581 so_30209962_30210182_2 . . . 160
6.582 so_47190568_47191759_1 . . . 160
6.583 so_31016784_31016833_3 . . . 160
6.584 so_15941964_15941964_0 . . . 160

7 Declarations **162**
7.1 so_5487849_5488100_0 . . . 163
7.2 so_27983074_27983235_1 . . . 163
7.3 so_4690886_4690944_0 . . . 163
7.4 so_25118191_25118838_0 . . . 163
7.5 so_33273550_33273777_1 . . . 163
7.6 so_27081344_27081436_0 . . . 163
7.7 so_4335024_4337554_0 . . . 163
7.8 so_13138605_13139393_0 . . . 163
7.9 so_13138605_13139363_0 . . . 163
7.10 so_27768899_27768972_1 . . . 163
7.11 so_29433293_29433534_1 . . . 163
7.12 so_29433293_29433459_0 . . . 163
7.13 so_29433293_29433534_0 . . . 163
7.14 so_11048827_11048906_1 . . . 163
7.15 so_28667924_28701901_2 . . . 163

7.16 so_44345121_44345121_0 . . . 163
7.17 so_18352385_18353789_1 . . . 163
7.18 so_38304847_38316977_2 . . . 163
7.19 so_31804618_31804848_0 . . . 163
7.20 so_27343811_27343811_2 . . . 163
7.21 so_27343811_27343811_1 . . . 163
7.22 so_23015482_23015727_0 . . . 163
7.23 so_38934879_38935340_4 . . . 163
7.24 so_29460818_29460818_0 . . . 163
7.25 so_33197817_33198169_1 . . . 163
7.26 so_6399898_6399998_42 . . . 163
7.27 so_5129312_5129346_0 . . . 163
7.28 so_29433293_29433534_3 . . . 163
7.29 so_23345554_23346104_1 . . . 163
7.30 so_31535401_31535509_1 . . . 163
7.31 so_27637768_27638059_0 . . . 163
7.32 so_15604411_15604573_10 . . . 163
7.33 so_21026264_21026358_5 . . . 163
7.34 so_24577997_24578063_0 . . . 163
7.35 so_37873874_37874421_1 . . . 164
7.36 so_37873874_37874305_0 . . . 164
7.37 so_37873874_37874305_1 . . . 164
7.38 so_37873874_37874421_2 . . . 164
7.39 so_27081344_27081436_1 . . . 164
7.40 Specifiers . . . 164
7.41 so_34502044_34502193_2 . . . 164
7.42 so_34450095_34994058_2 . . . 164
7.43 so_34450095_34994058_0 . . . 164
7.44 so_34450095_34994058_3 . . . 164
7.45 so_36113229_36113229_1 . . . 164
7.46 so_36113229_36113229_0 . . . 164
7.47 so_37037842_37037842_1 . . . 164
7.48 so_37037842_37037842_0 . . . 164
7.49 so_11779704_11779961_0 . . . 164
7.50 so_25788127_31652888_0 . . . 165
7.51 so_27983074_27983235_2 . . . 165
7.52 so_37879642_37882570_0 . . . 165
7.53 so_41019149_41019327_0 . . . 165
7.54 so_31536805_31540251_0 . . . 165
7.55 so_30999381_30999869_0 . . . 165
7.56 so_46752714_46763686_0 . . . 165
7.57 so_46752714_46753679_1 . . . 165
7.58 so_38176000_38176030_0 . . . 165
7.59 so_10675072_10675122_0 . . . 165
7.60 so_17559774_17560078_0 . . . 165
7.61 so_30734351_30734604_0 . . . 165
7.62 so_41942156_41942235_0 . . . 165
7.63 so_41942156_41942340_0 . . . 165
7.64 so_15908835_15908876_1 . . . 165
7.65 so_1256246_1256273_1 . . . 165
7.66 so_1256246_1256293_0 . . . 165

7.67 so_10675072_10675122_1 . . . 165
7.68 so_10675072_10675111_0 . . . 165
7.69 so_11880100_11880216_0 . . . 165
7.70 so_17559774_17559955_2 . . . 165
7.71 so_17559774_17559963_0 . . . 165
7.72 so_17559774_17560078_1 . . . 165
7.73 so_17471525_17472621_2 . . . 165
7.74 so_44788412_44788867_0 . . . 165
7.75 so_28707388_28707520_0 . . . 165
7.76 so_40753597_40753678_0 . . . 165
7.77 so_14422527_14422594_0 . . . 165
7.78 so_38932527_38932720_0 . . . 165
7.79 so_21026264_21026358_0 . . . 165
7.80 so_1943847_1943871_1 . . . 165
7.81 so_26085345_26085557_0 . . . 165
7.82 so_27541028_27541143_3 . . . 165
7.83 so_40419244_40419484_0 . . . 165
7.84 so_12256440_12256440_0 . . . 165
7.85 so_12256440_12256452_0 . . . 165
7.86 so_14403796_14404206_1 . . . 165
7.87 so_5261299_5261430_0 . . . 165
7.88 so_9241255_9241363_0 . . . 166
7.89 so_25853258_25854076_0 . . . 166
7.90 so_8952593_8952663_1 . . . 166
7.91 so_8576509_8576705_2 . . . 167
7.92 so_27398049_27398497_0 . . . 167
7.93 so_24942339_24942417_0 . . . 167
7.94 so_18586189_18586708_4 . . . 167
7.95 so_33118802_33119199_0 . . . 167
7.96 so_5388097_5388135_0 . . . 167
7.97 so_14775074_14775182_1 . . . 167
7.98 so_2501776_2501873_0 . . . 167
7.99 so_17536196_17536450_3 . . . 167
7.100 so_44788412_44788867_1 . . . 167
7.101 so_8392885_8393102_1 . . . 167
7.102 so_8392885_8393068_1 . . . 167
7.103 so_25284863_25286368_0 . . . 167
7.104 so_603390_603400_7 . . . 167
7.105 so_32299171_32299536_1 . . . 167
7.106 so_23669716_23671261_0 . . . 167
7.107 so_27835004_27835102_0 . . . 167
7.108 so_9436969_9437119_0 . . . 167
7.109 so_1932311_1932371_3 . . . 167
7.110 so_14269604_14269639_1 . . . 167
7.111 so_31812723_31813156_1 . . . 167
7.112 so_3647053_3647130_2 . . . 167
7.113 so_37093005_37093026_0 . . . 167
7.114 so_40658617_40658995_1 . . . 167
7.115 so_14036366_14036459_0 . . . 167
7.116 so_14036366_14036459_1 . . . 167
7.117 so_1611593_1612402_0 . . . 167

7.118 so_21632883_21633032_1 . . . 167
7.119 so_13375400_13375689_1 . . . 167
7.120 so_29871138_29957648_0 . . . 167
7.121 so_18202324_18202615_1 . . . 167
7.122 so_185624_189162_5 . . . 167
7.123 so_40258528_40269472_0 . . . 167
7.124 so_10259665_10262415_4 . . . 167
7.125 so_24644982_24645738_0 . . . 167
7.126 so_14775074_14775182_2 . . . 167
7.127 so_2501776_2501873_1 . . . 167
7.128 so_26054494_26054719_0 . . . 167
7.129 so_15865634_15865729_6 . . . 167
7.130 so_603390_603400_9 . . . 167
7.131 so_603390_603400_8 . . . 167
7.132 so_30030658_30030946_0 . . . 167
7.133 so_32736901_32737013_0 . . . 167
7.134 so_14797755_14797778_1 . . . 167
7.135 so_4866290_4866383_2 . . . 167
7.136 so_16530970_16530970_1 . . . 167
7.137 so_3342816_3343011_0 . . . 167
7.138 so_15960641_15960745_0 . . . 167
7.139 so_9734175_9734198_0 . . . 167
7.140 so_29871138_29957648_3 . . . 167
7.141 so_29871138_29957648_1 . . . 167
7.142 so_24104814_24105421_1 . . . 167
7.143 so_24104814_24105421_4 . . . 167
7.144 so_34110703_34110836_0 . . . 167
7.145 so_8611725_17692736_0 . . . 167
7.146 so_8611725_17692736_1 . . . 167
7.147 so_14775074_14775182_3 . . . 167
7.148 so_27642782_27642782_0 . . . 167
7.149 so_13299742_13299913_0 . . . 167
7.150 so_4769479_4769571_1 . . . 167
7.151 so_4769479_4769571_0 . . . 167
7.152 so_10847176_10847368_6 . . . 167
7.153 so_24716801_24716833_2 . . . 167
7.154 so_27605629_27605757_1 . . . 167
7.155 so_27605629_27605757_0 . . . 167
7.156 so_27605629_27605763_0 . . . 167
7.157 so_22016495_22032011_1 . . . 167
7.158 so_22016495_22032011_6 . . . 167
7.159 so_22016495_22032011_9 . . . 167
7.160 so_22016495_22032011_2 . . . 167
7.161 so_2501776_2501873_2 . . . 167
7.162 so_41670800_41672338_0 . . . 167
7.163 so_25596656_25596789_1 . . . 167
7.164 so_26054494_26054719_1 . . . 167
7.165 so_17536196_17557626_0 . . . 167
7.166 so_17536196_17536450_1 . . . 167
7.167 so_17536196_17536450_2 . . . 167
7.168 so_17536196_17536450_4 . . . 167

7.169 so_13678906_13686243_1 . . . 167
7.170 so_15234572_15234667_1 . . . 167
7.171 so_9305381_9305381_0 . . . 167
7.172 so_8392885_8393068_2 . . . 167
7.173 so_7567629_7567719_0 . . . 167
7.174 so_603390_603400_10 . . . 167
7.175 so_603390_603400_4 . . . 167
7.176 so_42932453_42933060_0 . . . 167
7.177 so_17054367_17054440_0 . . . 167
7.178 so_30455820_30456177_1 . . . 167
7.179 so_29287361_29287458_1 . . . 167
7.180 so_19858210_47382585_0 . . . 167
7.181 so_11428147_11428197_0 . . . 167
7.182 so_9648671_9648769_0 . . . 167
7.183 so_32172137_32172210_0 . . . 167
7.184 so_31495877_31798593_0 . . . 167
7.185 so_35826142_35826269_0 . . . 167
7.186 so_41491221_41493715_5 . . . 167
7.187 so_41491221_41491221_3 . . . 167
7.188 so_26307679_26307924_0 . . . 167
7.189 so_33533944_33534067_0 . . . 167
7.190 so_9853318_9853337_0 . . . 167
7.191 so_9853318_9853556_0 . . . 167
7.192 so_24384412_24384629_0 . . . 167
7.193 so_20000856_20002356_0 . . . 167
7.194 so_3647053_3647130_5 . . . 167
7.195 so_3647053_3647130_6 . . . 167
7.196 so_28348900_28349325_0 . . . 167
7.197 so_30309296_30309577_0 . . . 167
7.198 so_19134124_19134422_2 . . . 167
7.199 so_18860895_42069703_2 . . . 167
7.200 so_4866290_4866383_0 . . . 167
7.201 so_20014536_20014588_0 . . . 167
7.202 so_35398013_35398134_1 . . . 167
7.203 so_16760983_16761002_3 . . . 167
7.204 so_24170568_24170568_2 . . . 167
7.205 so_1389337_1389403_0 . . . 167
7.206 so_908830_910686_1 . . . 167
7.207 so_34717823_34721371_3 . . . 167
7.208 so_20730226_20730292_0 . . . 167
7.209 so_16219711_16219760_2 . . . 167
7.210 so_36571061_36571636_0 . . . 167
7.211 so_32879038_32879038_1 . . . 167
7.212 so_2138110_2138181_0 . . . 167
7.213 so_31791103_31791236_8 . . . 167
7.214 so_29660706_29660804_0 . . . 167
7.215 so_18629184_18629404_2 . . . 167
7.216 so_18629184_18629454_9 . . . 167
7.217 so_43736349_43736480_0 . . . 167
7.218 so_27345284_27345851_1 . . . 167
7.219 so_21632883_21633032_2 . . . 167

7.220 so_10488176_10488285_0 . . . 167
7.221 so_9415075_9415539_0 . . . 167
7.222 so_18202324_18202615_3 . . . 167
7.223 so_24104814_24105421_3 . . . 167
7.224 so_24104814_24105421_6 . . . 167
7.225 so_185624_189162_7 . . . 167
7.226 so_185624_1384400_0 . . . 167
7.227 so_12305646_12305725_0 . . . 167
7.228 so_46726390_46742542_0 . . . 167
7.229 so_11200805_11200894_0 . . . 167
7.230 so_15599887_15599912_0 . . . 167
7.231 so_47201589_47201947_0 . . . 167
7.232 so_25797291_25797332_0 . . . 167
7.233 so_25797291_25797439_0 . . . 167
7.234 so_33118802_33119199_1 . . . 167
7.235 so_22597074_22597152_0 . . . 167
7.236 so_31269660_31269660_0 . . . 168
7.237 so_31269660_31270598_0 . . . 168
7.238 so_31269660_31270598_1 . . . 168
7.239 so_32938686_32938769_0 . . . 168
7.240 so_5939486_5939961_1 . . . 168
7.241 so_1329191_1329232_1 . . . 168
7.242 so_612328_612363_2 . . . 168
7.243 so_44337152_44337477_1 . . . 168
7.244 so_27573012_27573166_0 . . . 168
7.245 so_32848065_32848342_1 . . . 168
7.246 so_30155802_30156450_4 . . . 168
7.247 so_13139235_13139291_2 . . . 168
7.248 so_13139235_13139291_1 . . . 168
7.249 so_30641865_30641971_0 . . . 168
7.250 so_10103453_10103453_0 . . . 168
7.251 so_6399898_6399998_63 . . . 168
7.252 so_6399898_6399931_1 . . . 168
7.253 so_30552141_30552236_2 . . . 168
7.254 so_29834657_29834997_2 . . . 168
7.255 so_29834657_29834997_0 . . . 168
7.256 so_40964593_40964747_0 . . . 168
7.257 so_36885305_36885605_0 . . . 168
7.258 so_43787462_43792468_4 . . . 168
7.259 so_22061750_22061811_2 . . . 168
7.260 so_22061750_22061811_0 . . . 168
7.261 so_30650073_30654483_0 . . . 168
7.262 so_4464300_4464739_0 . . . 168
7.263 so_4464300_4464481_0 . . . 168
7.264 so_29433293_29433459_1 . . . 168
7.265 so_29133988_29136447_1 . . . 168
7.266 so_20790932_20791007_0 . . . 168
7.267 so_15788947_15912208_0 . . . 168
7.268 so_32938686_32938769_1 . . . 168
7.269 so_31370305_31370450_0 . . . 168
7.270 so_28246988_28247015_0 . . . 168

7.271 so_44337152_44337477_2 . . . 168
7.272 so_18800121_18800151_0 . . . 168
7.273 so_27573012_27573166_1 . . . 168
7.274 so_28974316_28974609_2 . . . 168
7.275 so_10747810_10748056_0 . . . 168
7.276 so_34811364_34811672_1 . . . 168
7.277 so_41167119_41200573_5 . . . 168
7.278 so_28247170_28247271_0 . . . 168
7.279 so_19438407_19438466_0 . . . 168
7.280 so_43787462_43792468_5 . . . 168
7.281 so_18287151_18287422_1 . . . 168
7.282 so_33212696_33213424_0 . . . 168
7.283 so_8594954_8595131_0 . . . 168
7.284 so_26431190_26431541_1 . . . 168
7.285 so_13674368_13674512_0 . . . 168
7.286 so_44484660_44485308_0 . . . 168
7.287 so_10740962_10741060_0 . . . 168
7.288 so_6708726_6708726_0 . . . 168
7.289 so_22739408_22739577_3 . . . 168
7.290 so_6399898_6399998_67 . . . 168
7.291 so_22386191_22386307_2 . . . 168
7.292 so_22386191_22386307_0 . . . 168
7.293 so_13649071_13649159_1 . . . 168
7.294 so_5231680_5231750_0 . . . 168
7.295 so_31903594_31903690_0 . . . 168
7.296 so_22386191_22386307_1 . . . 169
7.297 so_21966230_21968228_1 . . . 169
7.298 so_18160290_18160419_1 . . . 169
7.299 so_3368129_3368191_1 . . . 169
7.300 so_8326108_8326170_0 . . . 169
7.301 so_34036482_34118342_0 . . . 169
7.302 so_30155802_30156450_7 . . . 169
7.303 so_22739408_22739577_1 . . . 169
7.304 so_15191127_15191275_0 . . . 169
7.305 so_11336507_11373917_1 . . . 169
7.306 so_31903594_31903690_2 . . . 169
7.307 so_4182013_4182095_0 . . . 169
7.308 so_3368129_3368191_2 . . . 169
7.309 so_8326108_8326170_2 . . . 169
7.310 so_22386191_22386307_3 . . . 169
7.311 so_15124077_15124077_0 . . . 169
7.312 so_28495119_28495626_1 . . . 169
7.313 so_29336262_29336262_0 . . . 169
7.314 so_9146167_9146523_0 . . . 169
7.315 so_11478962_11479069_0 . . . 169
7.316 so_30381690_30382242_1 . . . 169
7.317 so_30381690_30382242_0 . . . 169
7.318 so_2823485_2824067_2 . . . 169
7.319 so_22386191_22386307_4 . . . 169
7.320 so_19060720_19061195_1 . . . 169
7.321 so_3865011_3865259_0 . . . 169

7.322 so_4464300_4464739_1 . . . 169
7.323 so_5939486_5942427_0 . . . 169
7.324 so_612328_612363_3 . . . 169
7.325 so_25898127_25898127_0 . . . 169
7.326 so_41167119_41200573_3 . . . 169
7.327 so_29133988_29136447_2 . . . 169
7.328 so_6399898_6399998_69 . . . 170
7.329 so_13907252_13912993_0 . . . 170
7.330 so_17074089_17074539_0 . . . 170
7.331 so_30831567_30831567_0 . . . 170
7.332 so_30208685_30208883_0 . . . 170
7.333 so_33993100_33999044_1 . . . 170
7.334 so_23404713_23404835_0 . . . 170
7.335 so_23404713_23404950_0 . . . 170
7.336 so_11928089_12558623_0 . . . 170
7.337 so_32174186_32174466_0 . . . 170
7.338 so_6420085_6420127_0 . . . 170
7.339 so_43172652_43174512_0 . . . 170
7.340 so_41622896_41622896_0 . . . 170
7.341 so_46726390_46742542_1 . . . 170
7.342 so_24716801_24716833_0 . . . 170
7.343 so_32299171_32299536_5 . . . 170
7.344 so_24152418_24361568_1 . . . 170
7.345 so_27303650_27303980_2 . . . 170
7.346 so_16719234_16719290_1 . . . 170
7.347 so_16219711_16219760_1 . . . 170
7.348 so_27345284_27345851_0 . . . 170
7.349 so_29871138_29957648_2 . . . 170
7.350 so_34835430_34835609_0 . . . 170
7.351 so_46696247_46696335_0 . . . 170
7.352 so_20478193_20478796_1 . . . 170
7.353 so_20478193_20478796_2 . . . 170
7.354 so_20478193_20478435_0 . . . 170
7.355 so_20478193_20478435_2 . . . 170
7.356 so_25152301_25152301_0 . . . 170
7.357 so_42687094_42687235_0 . . . 170
7.358 so_9255185_9255612_0 . . . 170
7.359 so_27903336_27904091_3 . . . 170
7.360 so_27903336_27904091_0 . . . 170
7.361 so_27903336_27904091_2 . . . 170
7.362 so_46756288_46757212_0 . . . 170
7.363 so_45349247_45350254_0 . . . 170
7.364 so_14085956_14086279_0 . . . 170
7.365 so_34555793_34558600_4 . . . 170
7.366 so_34555793_34558600_0 . . . 170
7.367 so_34555793_34558600_3 . . . 170
7.368 so_34555793_34558600_8 . . . 170
7.369 so_34555793_34558600_2 . . . 170
7.370 so_34828161_34828179_0 . . . 170
7.371 so_34280729_34280783_4 . . . 170
7.372 so_34280729_34280783_2 . . . 170

7.373 so_34280729_34280783_8 . . . 170
7.374 so_34280729_34280783_3 . . . 170
7.375 so_34280729_34280783_0 . . . 170
7.376 so_40548269_40548269_0 . . . 170
7.377 so_32699007_32699790_0 . . . 170
7.378 so_32699007_32699790_1 . . . 170
7.379 so_32699007_32699213_0 . . . 170
7.380 so_41509106_41509252_7 . . . 170
7.381 so_41509106_41509617_1 . . . 170
7.382 so_41509106_41509252_6 . . . 170
7.383 so_41509106_41509252_4 . . . 170
7.384 so_41509106_41509252_5 . . . 170
7.385 so_41509106_41509252_2 . . . 170
7.386 so_41509106_41509252_3 . . . 170
7.387 so_25941894_25941993_4 . . . 170
7.388 so_25941894_25941993_0 . . . 170
7.389 so_20810378_20810439_1 . . . 170
7.390 so_34272899_34272899_1 . . . 170
7.391 so_34272899_34272899_0 . . . 170
7.392 so_34211688_34214734_10 . . . 170
7.393 so_34211688_34214734_4 . . . 170
7.394 so_34211688_34214734_5 . . . 170
7.395 so_34211688_34214734_6 . . . 170
7.396 so_34211688_34214734_2 . . . 170
7.397 so_21806154_21806862_0 . . . 170
7.398 so_21806154_21806862_2 . . . 170
7.399 so_21806154_21806862_1 . . . 170
7.400 so_33095999_33096162_2 . . . 170
7.401 so_33095999_33096162_3 . . . 170
7.402 so_33095999_33096162_0 . . . 170
7.403 so_32009602_32010338_0 . . . 170
7.404 so_46780657_46780860_2 . . . 170
7.405 so_46780657_46780860_0 . . . 170
7.406 so_33813929_33813930_13 . . . 171
7.407 so_36342296_36342828_0 . . . 171
7.408 so_36342296_36342828_1 . . . 171
7.409 so_36342296_36342296_0 . . . 171
7.410 so_37413741_37415910_0 . . . 171
7.411 so_37413741_37415910_1 . . . 171
7.412 so_37413741_37415910_2 . . . 171
7.413 so_31766613_31766613_0 . . . 171
7.414 so_25941894_25941993_6 . . . 171
7.415 so_25941894_25941993_8 . . . 171
7.416 so_25941894_25941993_5 . . . 171
7.417 so_25941894_25941993_7 . . . 171
7.418 so_25941894_25941993_10 . . . 171
7.419 so_25941894_25941993_9 . . . 171
7.420 so_32489702_32490017_2 . . . 171
7.421 so_32489702_32490017_6 . . . 171
7.422 so_32489702_32490017_10 . . . 171
7.423 so_32489702_32490017_5 . . . 171

7.424 so_32489702_32490017_8 . . . 171
7.425 so_32489702_32490017_7 . . . 171
7.426 so_32489702_32490017_9 . . . 171
7.427 so_32489702_32490017_4 . . . 171
7.428 so_32489702_32490017_0 . . . 171
7.429 so_20744526_20744647_0 . . . 171
7.430 so_20744526_20744647_2 . . . 171
7.431 so_45025808_45025808_0 . . . 172
7.432 so_22176777_22177425_0 . . . 172
7.433 so_37866375_37866503_1 . . . 172
7.434 so_29900218_29900558_2 . . . 172
7.435 so_29900218_29900558_1 . . . 172
7.436 so_41961181_41961181_0 . . . 172
7.437 so_45268954_45269409_1 . . . 172
7.438 so_39723288_39723288_2 . . . 172
7.439 so_39723288_39723288_0 . . . 172
7.440 so_39723288_39723532_0 . . . 172
7.441 so_39723288_39723288_1 . . . 172
7.442 so_19126609_19126975_0 . . . 172
7.443 so_46253990_46261649_3 . . . 172
7.444 so_29577778_29577884_0 . . . 172
7.445 so_13038985_13039987_0 . . . 172
7.446 so_26529542_26530069_0 . . . 172
7.447 so_34555793_34558600_9 . . . 172
7.448 so_31903037_31903244_0 . . . 172
7.449 so_23775705_23775840_2 . . . 172
7.450 so_31583326_31583846_1 . . . 172
7.451 so_31583326_31583846_2 . . . 172
7.452 so_34280729_34280783_9 . . . 172
7.453 so_38851469_38853356_2 . . . 172
7.454 so_13295089_14472252_0 . . . 172
7.455 so_21897015_21897696_0 . . . 172
7.456 so_31683519_31683519_0 . . . 172
7.457 so_35659155_35659573_1 . . . 172
7.458 so_32803708_32804644_0 . . . 172
7.459 so_34272899_34272949_0 . . . 172
7.460 so_34211688_34214734_12 . . . 172
7.461 so_31206937_31206937_0 . . . 172
7.462 so_40322579_40324901_0 . . . 172
7.463 so_20478193_20478796_3 . . . 172
7.464 so_5605142_5612606_2 . . . 172
7.465 so_36430134_36430425_0 . . . 172
7.466 so_46215910_46217058_0 . . . 172
7.467 so_40548269_40553112_0 . . . 172
7.468 so_27680413_27680626_1 . . . 172
7.469 so_32009602_32010338_1 . . . 172
7.470 so_11522399_11523155_1 . . . 172
7.471 so_11522399_11800201_1 . . . 172
7.472 so_26568820_26598013_0 . . . 172
7.473 so_20437734_20448518_0 . . . 172
7.474 so_9152580_9152753_0 . . . 172

7.475 so_46253990_46261649_0 . . . 172
7.476 so_46253990_46261649_1 . . . 172
7.477 so_13038985_13039987_1 . . . 172
7.478 so_28880538_28880915_0 . . . 172
7.479 so_12785691_12785734_0 . . . 172
7.480 so_30089131_30089692_0 . . . 172
7.481 so_31410933_31410986_2 . . . 172
7.482 so_37993732_37994483_0 . . . 172
7.483 so_28883510_28883895_0 . . . 173
7.484 so_31735037_31736000_0 . . . 173
7.485 so_28614591_28614785_1 . . . 173
7.486 so_34240136_34240330_0 . . . 173
7.487 so_29432283_29432462_2 . . . 173
7.488 so_28755515_28755515_0 . . . 173
7.489 so_16057690_16088912_1 . . . 173
7.490 so_13571749_13784100_2 . . . 173
7.491 so_14616183_14616201_1 . . . 173
7.492 so_42746099_42746285_1 . . . 173
7.493 so_30527865_30528095_0 . . . 173
7.494 so_16063123_16063123_0 . . . 173
7.495 so_33813929_33813930_0 . . . 173
7.496 so_24342455_24342841_3 . . . 173
7.497 so_24342455_24342841_2 . . . 173
7.498 so_13104358_13104505_1 . . . 173
7.499 so_13104358_13104492_0 . . . 173
7.500 so_37932066_37932254_1 . . . 173
7.501 so_29255322_29255535_2 . . . 173
7.502 so_43535233_43535233_0 . . . 173
7.503 so_27303650_27303980_3 . . . 173
7.504 so_23404713_23404835_1 . . . 173
7.505 so_33197817_33198169_0 . . . 173
7.506 so_33197817_33198182_0 . . . 173
7.507 so_27221883_27222359_4 . . . 173
7.508 so_17480440_17480596_0 . . . 173
7.509 so_29307521_29307679_1 . . . 173
7.510 so_31766613_31766689_2 . . . 173
7.511 so_40176585_40176800_0 . . . 173
7.512 so_46208655_46208860_3 . . . 173
7.513 so_27744079_27744079_0 . . . 173
7.514 so_43172652_43174512_1 . . . 173
7.515 so_30172483_30172740_0 . . . 173
7.516 so_31519935_31519935_0 . . . 173
7.517 so_11927032_11927364_0 . . . 173
7.518 so_4999241_4999241_0 . . . 173
7.519 so_20870233_20870343_0 . . . 173
7.520 so_31269660_31270540_6 . . . 173
7.521 so_31269660_31270540_2 . . . 173
7.522 so_31269660_31270540_3 . . . 173
7.523 so_31269660_31270540_5 . . . 173
7.524 so_31269660_31270540_0 . . . 173
7.525 so_31269660_31269660_1 . . . 173

7.526 so_31269660_31270540_4 . . . 173
7.527 so_31269660_31270540_1 . . . 173
7.528 so_28755515_28755515_1 . . . 173
7.529 so_11534126_11534268_0 . . . 173
7.530 so_11534126_11534268_1 . . . 173
7.531 so_13808932_13808933_7 . . . 173
7.532 so_30798661_30798876_0 . . . 173
7.533 so_30798661_30799045_0 . . . 173
7.534 so_30798661_30798876_2 . . . 173
7.535 so_30798661_30798885_1 . . . 173
7.536 so_5781222_5781284_1 . . . 173
7.537 so_5781222_5781284_6 . . . 173
7.538 so_5781222_5781284_5 . . . 173
7.539 so_5781222_5781284_4 . . . 173
7.540 so_5781222_5781284_2 . . . 173
7.541 so_5781222_5781284_0 . . . 173
7.542 so_5781222_5781284_3 . . . 173
7.543 so_27432506_27432506_6 . . . 173
7.544 so_27432506_27432506_9 . . . 173
7.545 so_27432506_27432506_1 . . . 173
7.546 so_27432506_27432506_10 . . . 173
7.547 so_27432506_27432506_4 . . . 173
7.548 so_27432506_27432506_5 . . . 173
7.549 so_27432506_27432506_11 . . . 173
7.550 so_27432506_27432506_8 . . . 173
7.551 so_27432506_27432506_2 . . . 173
7.552 so_27432506_27432506_3 . . . 173
7.553 so_27432506_27432506_7 . . . 173
7.554 so_27432506_27432506_13 . . . 173
7.555 so_27432506_27432506_12 . . . 173
7.556 so_37166131_37166556_5 . . . 173
7.557 so_20551632_20551660_1 . . . 173
7.558 so_20551632_20551660_3 . . . 173
7.559 so_36157337_36157694_0 . . . 173
7.560 so_36157337_36157694_1 . . . 173
7.561 so_32856481_32856568_3 . . . 173
7.562 so_32856481_32856568_1 . . . 173
7.563 so_28311795_28311795_5 . . . 173
7.564 so_28495119_28495626_0 . . . 174
7.565 so_4573090_4573103_0 . . . 174
7.566 so_20980192_20981414_3 . . . 174
7.567 so_20980192_20981408_0 . . . 174
7.568 so_45478054_45478126_5 . . . 174
7.569 so_32641531_32642696_1 . . . 174
7.570 so_41883379_41883474_1 . . . 174
7.571 so_11534126_11534268_2 . . . 174
7.572 so_27983074_27983235_3 . . . 174
7.573 so_26184441_26184684_0 . . . 174
7.574 so_33263625_33263995_0 . . . 174
7.575 so_30798661_30798774_0 . . . 174
7.576 so_30798661_30799045_1 . . . 174

7.577 so_30798661_30798876_3 . . . 174
7.578 so_5781222_5781308_2 . . . 174
7.579 so_16856596_16856772_0 . . . 174
7.580 so_27768899_27768899_0 . . . 174
7.581 so_20909693_20915412_0 . . . 174
7.582 so_42251142_42251142_0 . . . 174
7.583 so_23629856_23630403_0 . . . 174
7.584 so_21894357_21895421_0 . . . 174
7.585 so_24433250_24433518_1 . . . 174
7.586 so_11077566_11077786_0 . . . 174
7.587 so_3484233_3484265_4 . . . 174
7.588 so_3484233_3485333_0 . . . 174
7.589 so_27195716_27195716_0 . . . 174
7.590 so_25460275_25460987_0 . . . 174
7.591 so_1789768_1789808_0 . . . 174
7.592 so_36670354_36670545_1 . . . 174
7.593 so_7598152_7598179_0 . . . 174
7.594 so_27297034_27297034_0 . . . 174
7.595 so_40592631_40592751_0 . . . 174
7.596 so_712334_712534_0 . . . 174
7.597 so_23754929_23755109_1 . . . 174
7.598 so_21529194_21529283_1 . . . 174
7.599 so_26291427_26291561_0 . . . 174
7.600 so_22656734_22656791_0 . . . 174
7.601 so_26128980_26129015_0 . . . 174
7.602 so_40796662_40967442_1 . . . 174
7.603 so_212237_212693_0 . . . 174
7.604 so_25345511_25345538_0 . . . 174
7.605 so_46761431_46761456_0 . . . 174
7.606 so_9890218_9890859_0 . . . 174
7.607 so_1934367_1934456_0 . . . 174
7.608 so_21509523_21509530_0 . . . 174
7.609 so_37751330_37751443_0 . . . 174
7.610 so_35666354_35669885_0 . . . 174
7.611 so_40294063_40294459_0 . . . 174
7.612 so_20602859_20602929_0 . . . 174
7.613 so_14040995_14041323_0 . . . 174
7.614 so_25209838_25209904_0 . . . 174
7.615 so_16505934_16505946_1 . . . 174
7.616 so_24433250_24433518_2 . . . 174
7.617 so_583076_583150_0 . . . 174
7.618 so_11091385_11091445_1 . . . 174
7.619 so_11091385_11091552_0 . . . 174
7.620 so_21156509_21156631_0 . . . 174
7.621 so_11077566_11077786_1 . . . 174
7.622 so_27221883_27222359_3 . . . 174
7.623 so_3484233_3485333_1 . . . 174
7.624 so_3484233_3484265_5 . . . 174
7.625 so_17290376_17290493_1 . . . 174
7.626 so_17290376_17290474_0 . . . 174
7.627 so_9841641_9842103_0 . . . 174

7.628 so_37708544_37708544_0 . . . 174
7.629 so_2271046_2271687_0 . . . 174
7.630 so_43417462_43417710_0 . . . 174
7.631 so_22808067_22808068_3 . . . 174
7.632 so_8936549_8936637_0 . . . 174
7.633 so_29883327_29883410_0 . . . 174
7.634 so_29883327_29883695_1 . . . 174
7.635 so_15636100_15636397_0 . . . 174
7.636 so_18586189_18586708_7 . . . 174
7.637 so_16089722_16089757_0 . . . 174
7.638 so_19938365_19939597_0 . . . 174
7.639 so_44870557_44871311_0 . . . 174
7.640 so_10792757_10792766_0 . . . 174
7.641 so_32070395_32071868_1 . . . 174
7.642 so_16256065_16256081_1 . . . 174
7.643 so_10906595_10906639_1 . . . 174
7.644 so_1789768_1789808_1 . . . 174
7.645 so_29344299_29344563_0 . . . 174
7.646 so_6987497_6987704_1 . . . 174
7.647 so_8663290_8663397_1 . . . 174
7.648 so_2508605_43773764_0 . . . 174
7.649 so_25406818_25406977_1 . . . 174
7.650 so_16802701_16802727_0 . . . 174
7.651 so_7367580_7368038_2 . . . 175
7.652 so_7367580_7368038_3 . . . 175
7.653 so_25630662_25630837_0 . . . 175
7.654 so_31527499_31527581_0 . . . 175
7.655 so_31527499_31527581_1 . . . 175
7.656 so_42233379_42233848_1 . . . 175
7.657 so_41096733_41097201_0 . . . 175
7.658 so_25944309_25945302_0 . . . 175
7.659 so_2491495_4875393_0 . . . 175
7.660 so_32106803_32107221_1 . . . 175
7.661 so_24555288_24555400_1 . . . 175
7.662 so_21156509_21156967_0 . . . 175
7.663 so_21156509_21156631_1 . . . 175
7.664 so_16732788_16733903_0 . . . 175
7.665 so_29868792_29868986_0 . . . 175
7.666 so_8049013_8049102_0 . . . 175
7.667 so_18550784_18550834_0 . . . 175
7.668 so_3263873_19675409_1 . . . 175
7.669 so_4437527_4437555_1 . . . 175
7.670 so_12710336_12710360_1 . . . 175
7.671 so_25944309_25945302_1 . . . 175
7.672 so_11782973_11783200_4 . . . 175
7.673 so_21065642_21065709_0 . . . 175
7.674 so_33280881_33281221_0 . . . 175
7.675 so_33684826_33684826_0 . . . 175
7.676 so_4479597_4479652_0 . . . 175
7.677 so_36657121_36657277_3 . . . 175
7.678 so_47253655_47253655_0 . . . 175

7.679 so_46429692_46429926_1 . . . 175
7.680 so_32519082_32519417_0 . . . 176
7.681 so_2432111_2435557_0 . . . 176
7.682 so_38977808_38978223_2 . . . 176
7.683 so_38977808_38978223_1 . . . 176
7.684 so_42608492_42608492_0 . . . 176
7.685 so_23202296_23202480_1 . . . 176
7.686 so_19640187_19640743_0 . . . 176
7.687 so_44188769_44188888_0 . . . 176
7.688 so_32856481_32856568_4 . . . 176
7.689 so_22493879_22494020_2 . . . 176
7.690 so_22493879_22494119_5 . . . 176
7.691 so_22493879_22494119_7 . . . 176
7.692 so_22493879_22494020_1 . . . 176
7.693 so_22493879_22494020_4 . . . 176
7.694 so_22493879_22494119_9 . . . 176
7.695 so_28153959_28154026_0 . . . 176
7.696 so_2144012_2145942_0 . . . 176
7.697 so_2144012_2434989_1 . . . 176
7.698 so_5539249_37438120_4 . . . 176
7.699 so_21144541_21144887_0 . . . 176
7.700 so_472530_472530_1 . . . 176
7.701 so_13810222_13810408_0 . . . 176
7.702 so_30726423_30726614_1 . . . 176
7.703 so_30726423_30726896_0 . . . 176
7.704 so_23202296_23202480_2 . . . 176
7.705 so_13485256_13485394_0 . . . 176
7.706 so_37482246_37593205_0 . . . 176
7.707 so_3097779_3097803_0 . . . 178
7.708 so_36517596_36517737_0 . . . 178
7.709 so_15899369_15899369_0 . . . 178
7.710 so_15899369_15899369_1 . . . 178
7.711 so_26199545_26199701_1 . . . 178
7.712 so_33691417_33714950_3 . . . 178
7.713 so_33691417_33714950_2 . . . 178
7.714 so_33691417_33714950_1 . . . 178
7.715 so_27557369_27557450_0 . . . 178
7.716 so_23191702_23193909_0 . . . 178
7.717 so_20021292_20021356_1 . . . 178
7.718 so_28265984_28266104_2 . . . 178
7.719 so_28265984_28266104_4 . . . 178
7.720 so_28265984_28266104_3 . . . 178
7.721 so_28265984_28266104_1 . . . 178
7.722 so_37312668_37313013_3 . . . 178
7.723 so_10359590_10359808_1 . . . 178
7.724 so_10359590_10359808_0 . . . 178
7.725 so_38371924_38373019_2 . . . 178
7.726 so_42046021_42046360_1 . . . 178
7.727 so_40605343_40606884_0 . . . 178
7.728 so_37167398_37167521_0 . . . 178
7.729 so_37167398_37167505_0 . . . 178

7.730 so_24121708_24122429_9 . . . 178
7.731 so_24121708_24122429_10 . . . 178
7.732 so_24121708_24122429_8 . . . 178
7.733 so_33479631_33480118_1 . . . 178
7.734 so_33479631_33480126_0 . . . 178
7.735 so_33479631_33480118_2 . . . 178
7.736 so_33479631_33480152_0 . . . 178
7.737 so_33479631_33479631_0 . . . 178
7.738 so_33479631_33480118_3 . . . 178
7.739 so_37185803_37189273_7 . . . 178
7.740 so_37185803_37189273_6 . . . 178
7.741 so_37185803_37189273_11 . . . 178
7.742 so_37185803_37189273_9 . . . 178
7.743 so_37185803_37189273_8 . . . 178
7.744 so_25312225_25312375_0 . . . 178
7.745 so_25312225_25312375_1 . . . 178
7.746 so_25312225_25312375_3 . . . 178
7.747 so_25312225_25312375_2 . . . 178
7.748 so_25186601_25187224_0 . . . 178
7.749 so_27159187_27160109_1 . . . 178
7.750 so_26088799_26088799_3 . . . 178
7.751 so_26088799_26088799_1 . . . 178
7.752 so_26088799_26088799_2 . . . 178
7.753 so_26088799_26088799_4 . . . 178
7.754 so_24116817_24116818_12 . . . 178
7.755 so_24116817_24116818_8 . . . 178
7.756 so_24116817_24116818_10 . . . 178
7.757 so_24116817_24116818_9 . . . 178
7.758 so_24116817_24116818_7 . . . 178
7.759 so_42026327_42027557_2 . . . 178
7.760 so_32240444_32243289_0 . . . 178
7.761 so_32240444_32243289_2 . . . 178
7.762 so_15047277_15047278_3 . . . 178
7.763 so_15047277_15047278_1 . . . 178
7.764 so_15047277_15047278_2 . . . 178
7.765 so_15047277_15047278_4 . . . 178
7.766 so_35543211_35543211_0 . . . 178
7.767 so_32231075_32231125_0 . . . 178
7.768 so_42037687_42037687_1 . . . 178
7.769 so_27810202_27810260_1 . . . 178
7.770 so_26435084_26435208_0 . . . 178
7.771 so_33729176_33729446_3 . . . 178
7.772 so_33729176_33729446_1 . . . 178
7.773 so_33729176_33729446_4 . . . 178
7.774 so_33729176_33729446_2 . . . 178
7.775 so_33600951_33601014_1 . . . 178
7.776 so_33600951_33601014_0 . . . 178
7.777 so_33600951_33601014_2 . . . 178
7.778 so_46020637_46025291_0 . . . 178
7.779 so_41996441_41996580_2 . . . 178
7.780 so_12689870_12689935_0 . . . 178

7.781 so_3909720_3909825_0 . . . 178
7.782 so_42363040_42363605_6 . . . 178
7.783 so_42363040_42363605_5 . . . 178
7.784 so_42363040_42363605_4 . . . 178
7.785 so_42363040_42363605_7 . . . 178
7.786 so_25248680_25249142_0 . . . 178
7.787 so_44294743_44413900_3 . . . 178
7.788 so_44294743_44413900_2 . . . 178
7.789 so_44294743_44413900_4 . . . 178
7.790 so_17241614_17242295_16 . . . 178
7.791 so_17241614_17242295_2 . . . 178
7.792 so_17241614_17242295_9 . . . 178
7.793 so_17241614_17242295_3 . . . 178
7.794 so_17241614_17242295_5 . . . 178
7.795 so_17241614_17242295_7 . . . 178
7.796 so_17241614_17242295_8 . . . 178
7.797 so_17241614_17242295_4 . . . 178
7.798 so_30293262_30293277_1 . . . 178
7.799 so_35088599_35088909_4 . . . 178
7.800 so_38373768_38374157_6 . . . 178
7.801 so_38373768_38374157_3 . . . 178
7.802 so_38373768_38374157_0 . . . 178
7.803 so_38373768_38374157_4 . . . 178
7.804 so_38373768_38374157_2 . . . 178
7.805 so_3158999_3430035_0 . . . 178
7.806 so_3158999_3430035_3 . . . 178
7.807 so_3158999_3430035_1 . . . 178
7.808 so_5924870_5924870_0 . . . 178
7.809 so_37897810_37898353_0 . . . 179
7.810 so_8166018_8166480_0 . . . 179
7.811 so_7900284_7900484_0 . . . 179
7.812 so_39875499_39875640_1 . . . 179
7.813 so_4891335_4891681_0 . . . 179
7.814 so_34857135_34857135_0 . . . 179
7.815 so_9785912_9786892_3 . . . 179
7.816 so_9785912_9786892_1 . . . 179
7.817 so_9785912_9786892_4 . . . 179
7.818 so_21952658_21952658_0 . . . 179
7.819 so_30381690_30382259_0 . . . 179
7.820 so_6321191_6321326_0 . . . 179
7.821 so_25118191_25118838_1 . . . 179
7.822 so_702650_702701_0 . . . 179
7.823 so_9148329_9148358_0 . . . 179
7.824 so_8166018_8166480_1 . . . 179
7.825 so_26870724_26871006_1 . . . 179
7.826 so_6485419_6485521_0 . . . 179
7.827 so_9369845_9370714_1 . . . 179
7.828 so_22016378_22017286_0 . . . 179
7.829 so_32159000_32159584_0 . . . 179
7.830 so_28956250_28956470_0 . . . 179
7.831 so_44057294_44057312_1 . . . 179

7.832 so_31559264_31559661_0 . . . 179
7.833 so_45502231_45502531_0 . . . 180
7.834 so_40841777_40993650_0 . . . 180
7.835 so_46027276_46027318_1 . . . 180
7.836 so_24109737_24109800_0 . . . 180
7.837 so_32322255_32324037_2 . . . 180
7.838 so_24259947_24260964_0 . . . 180
7.839 so_32173390_32173517_0 . . . 180
7.840 so_18931993_19325983_2 . . . 180
7.841 so_42744245_42744535_1 . . . 180
7.842 so_42744245_42744535_0 . . . 180
7.843 so_23345554_23346104_2 . . . 180
7.844 so_39106798_39107034_2 . . . 180
7.845 so_31535401_31535509_2 . . . 180
7.846 so_13158354_13158487_0 . . . 180
7.847 so_17231576_17231914_0 . . . 180
7.848 so_27746467_27746467_0 . . . 180
7.849 so_46494719_46494893_0 . . . 180
7.850 so_24109737_24109800_1 . . . 180
7.851 so_27367400_27367692_0 . . . 180
7.852 so_41273101_41273286_1 . . . 180
7.853 so_37493769_37493769_1 . . . 180
7.854 so_24259947_24260964_1 . . . 180
7.855 so_32889492_32892157_0 . . . 180
7.856 so_14285198_14285294_2 . . . 180
7.857 so_18931993_19325983_3 . . . 180
7.858 so_18931993_19325983_1 . . . 180
7.859 so_45402604_45404450_0 . . . 180
7.860 so_44641253_44653025_0 . . . 180
7.861 so_17231576_17231914_1 . . . 180
7.862 so_42462502_42463942_0 . . . 180
7.863 so_32322255_32324037_1 . . . 180
7.864 so_28013106_28013314_0 . . . 180
7.865 so_17231576_17231576_0 . . . 180
7.866 so_17231576_17231914_2 . . . 180
7.867 so_43001428_43001428_1 . . . 180
7.868 so_26386010_26386337_1 . . . 180
7.869 so_14285198_14285294_3 . . . 180
7.870 so_32173390_32173517_1 . . . 180
7.871 so_24018932_24019645_4 . . . 180
7.872 so_24018932_24019645_3 . . . 180
7.873 so_11438341_11438514_0 . . . 180
7.874 so_17231576_17231914_4 . . . 180
7.875 so_43001428_43001428_0 . . . 180
7.876 so_4466433_4468520_1 . . . 180
7.877 so_12834874_12846168_0 . . . 180
7.878 so_18199728_18199974_0 . . . 180
7.879 so_14285198_14285294_4 . . . 180
7.880 so_11302981_11303175_0 . . . 180
7.881 so_2847734_2847768_0 . . . 180
7.882 so_37407547_37408242_0 . . . 180

7.883 so_17231576_17231914_5 . . . 180
7.884 so_42462502_42463942_1 . . . 180
7.885 so_46910844_46911517_0 . . . 180
7.886 so_4466433_4468520_2 . . . 180
7.887 so_32322255_32322613_0 . . . 180
7.888 so_26386010_26386337_3 . . . 180
7.889 so_14285198_14285294_5 . . . 180
7.890 so_18931993_19325983_0 . . . 180
7.891 so_27047795_27048079_0 . . . 180
7.892 so_39106798_39107034_1 . . . 180
7.893 so_24233796_24233822_0 . . . 180
7.894 so_16692499_16692619_0 . . . 180
7.895 so_40172392_40173491_0 . . . 180
7.896 so_17496268_17496410_2 . . . 180
7.897 so_40841777_40989539_1 . . . 180
7.898 so_32282705_32282705_0 . . . 180
7.899 so_16269587_20875790_0 . . . 180
7.900 so_44698195_44701336_1 . . . 180
7.901 so_23968479_23968722_1 . . . 180
7.902 so_23968479_23968790_0 . . . 180
7.903 so_17551316_17552291_0 . . . 180
7.904 so_44946408_44947538_1 . . . 180
7.905 so_24121708_24122429_3 . . . 180
7.906 so_32089214_32089365_1 . . . 180
7.907 so_39354065_39355695_0 . . . 180
7.908 so_39354065_39354065_0 . . . 180
7.909 so_39354065_39355695_1 . . . 180
7.910 so_35361408_35361729_0 . . . 180
7.911 so_11279647_11281056_0 . . . 180
7.912 so_27629039_27629039_0 . . . 180
7.913 so_26951779_26953869_0 . . . 180
7.914 so_33479631_33480118_0 . . . 180
7.915 so_31392020_31392265_0 . . . 180
7.916 so_25179414_25184733_2 . . . 180
7.917 so_44902192_44902192_1 . . . 180
7.918 so_44902192_44902192_2 . . . 180
7.919 so_30245091_30245730_0 . . . 180
7.920 so_31872991_31873347_0 . . . 180
7.921 so_24152017_24152066_2 . . . 180
7.922 so_24152017_24152066_5 . . . 180
7.923 so_24152017_24152066_1 . . . 180
7.924 so_24152017_24152066_4 . . . 180
7.925 so_30227277_30230036_0 . . . 180
7.926 so_31672968_31672968_1 . . . 180
7.927 so_12224733_12224793_0 . . . 180
7.928 so_30240851_30241159_0 . . . 180
7.929 so_6443230_6444871_0 . . . 180
7.930 so_12242449_12242489_1 . . . 180
7.931 so_16827034_16827131_0 . . . 180
7.932 so_27199478_27199584_0 . . . 180
7.933 so_27199478_27199571_0 . . . 180

7.934 so_12016757_12016897_1 . . . 180
7.935 so_32130745_32130883_0 . . . 180
7.936 so_31628366_31628464_0 . . . 180
7.937 so_27047795_27048079_1 . . . 180
7.938 so_46933091_46933478_0 . . . 180
7.939 so_27766601_27766740_0 . . . 180
7.940 so_32515183_32515251_2 . . . 180
7.941 so_32515183_32515264_0 . . . 180
7.942 so_32515183_32515251_4 . . . 180
7.943 so_32778604_32778832_0 . . . 181
7.944 so_33414252_33414417_1 . . . 181
7.945 so_33414252_33414417_2 . . . 181
7.946 so_31672968_31673020_2 . . . 181
7.947 so_31672968_31673020_1 . . . 181
7.948 so_16642294_16642294_1 . . . 181
7.949 so_16642294_16642462_0 . . . 181
7.950 so_16642294_16642539_0 . . . 181
7.951 so_34215341_34215431_0 . . . 181
7.952 so_23968479_23968790_1 . . . 181
7.953 so_23968479_23968722_2 . . . 181
7.954 so_26692637_26692775_6 . . . 181
7.955 so_44501629_44501766_0 . . . 181
7.956 so_27814859_27815007_0 . . . 181
7.957 so_45402604_45404450_1 . . . 181
7.958 so_44641253_44653025_2 . . . 181
7.959 so_23968479_23968790_2 . . . 182
7.960 so_23968479_23968722_3 . . . 182
7.961 so_27367400_27367692_1 . . . 182
7.962 so_17134975_17135093_0 . . . 182
7.963 so_22236324_22236516_0 . . . 182
7.964 so_22040875_22041210_0 . . . 182
7.965 so_33949403_33950753_0 . . . 182
7.966 so_27732195_27734079_0 . . . 182
7.967 so_19875092_19908754_2 . . . 182
7.968 so_23968479_23968790_3 . . . 182
7.969 so_23968479_23968722_4 . . . 182
7.970 so_46460508_46469080_0 . . . 182
7.971 so_24115444_24115958_4 . . . 182
7.972 so_24115444_24115958_5 . . . 182
7.973 so_27746467_27747057_0 . . . 182
7.974 so_42321573_42321690_0 . . . 182
7.975 so_28389792_28390022_1 . . . 182
7.976 so_43054365_43054439_0 . . . 182
7.977 so_17268974_17269073_0 . . . 182
7.978 so_42466838_42488624_1 . . . 182
7.979 so_42466838_42488624_0 . . . 182
7.980 so_43052499_43054585_0 . . . 182
7.981 so_32889492_32892157_1 . . . 182
7.982 so_8582610_27735933_0 . . . 182
7.983 so_17608637_17616371_0 . . . 182
7.984 so_25693676_25693792_0 . . . 182

7.985 so_24036357_24043642_1 . . . 182
7.986 so_38325730_38326110_0 . . . 182
7.987 so_45584616_45585426_0 . . . 182
7.988 so_27629816_27630346_2 . . . 182
7.989 so_24115444_24115958_1 . . . 182
7.990 so_24115444_24115958_2 . . . 182
7.991 so_46494719_46494893_1 . . . 182
7.992 so_37882546_37884028_0 . . . 182
7.993 so_26386010_26386337_4 . . . 182
7.994 so_32889492_32892157_3 . . . 182
7.995 so_32889492_32890958_0 . . . 182
7.996 so_32173390_32173517_2 . . . 182
7.997 so_44294743_44413900_11 . . . 182
7.998 so_25341378_25341458_0 . . . 182
7.999 so_37726206_37726704_0 . . . 183
7.1000 so_26268721_26268721_0 . . . 183
7.1001 Enumeration declarations . . . 183
7.1002 so_40018653_40020448_1 . . . 183
7.1003 so_25117677_25117860_0 . . . 184
7.1004 so_17045661_17046231_1 . . . 184
7.1005 so_34811486_34811571_0 . . . 184
7.1006 so_34811486_34811588_0 . . . 184
7.1007 so_34811486_34811578_0 . . . 184
7.1008 so_32504090_32504155_0 . . . 184
7.1009 so_6992593_6993152_0 . . . 184
7.1010 so_13464387_13464482_0 . . . 184
7.1011 so_13464387_13464486_0 . . . 184
7.1012 so_13464387_13464468_0 . . . 184
7.1013 so_6301575_6301619_1 . . . 184
7.1014 so_18752430_18752683_0 . . . 184
7.1015 so_18752430_18752504_1 . . . 184
7.1016 so_26838758_26838788_1 . . . 184
7.1017 so_26838758_26838788_0 . . . 184
7.1018 so_18868599_18868626_0 . . . 184
7.1019 so_40607013_40607125_0 . . . 184
7.1020 so_6492612_6492670_0 . . . 184
7.1021 so_11789434_11789434_0 . . . 184
7.1022 so_46244022_46244022_0 . . . 184
7.1023 so_23709019_23720793_5 . . . 184
7.1024 so_35998348_35998348_0 . . . 184
7.1025 so_8911188_8911596_0 . . . 184
7.1026 so_23730211_23745649_0 . . . 184
7.1027 so_14288713_14289291_0 . . . 184
7.1028 so_23341484_23342205_1 . . . 184
7.1029 so_27664057_27664057_0 . . . 184
7.1030 so_22621105_22621673_0 . . . 185
7.1031 so_17045661_17046802_1 . . . 185
7.1032 so_4665709_4665867_3 . . . 185
7.1033 so_41836658_41837472_1 . . . 185
7.1034 so_41836658_41837472_0 . . . 185
7.1035 so_41836658_41836952_0 . . . 185

7.1036 so_30852922_30853213_0 . . . 185
7.1037 so_27597163_27597163_0 . . . 185
7.1038 so_27597163_27597163_1 . . . 185
7.1039 so_27597163_27597163_3 . . . 185
7.1040 so_27597163_27597163_2 . . . 185
7.1041 so_27597163_27597304_0 . . . 185
7.1042 so_19423626_19424134_0 . . . 185
7.1043 so_37752002_37757636_1 . . . 185
7.1044 so_7676817_7676940_0 . . . 185
7.1045 so_32950709_32950795_0 . . . 185
7.1046 so_40395066_40437060_1 . . . 185
7.1047 so_43012458_43012547_0 . . . 185
7.1048 so_34376921_34377119_1 . . . 185
7.1049 so_23323510_23323510_1 . . . 185
7.1050 so_23323510_23323510_2 . . . 185
7.1051 so_9883939_9884173_0 . . . 185
7.1052 so_9883939_9884173_3 . . . 185
7.1053 so_28659804_28660475_0 . . . 185
7.1054 so_38347643_38347643_0 . . . 185
7.1055 so_38347643_38348173_0 . . . 185
7.1056 so_17888322_17888598_0 . . . 185
7.1057 so_38336578_38336740_1 . . . 185
7.1058 so_38336578_38336740_0 . . . 185
7.1059 so_33812998_33814842_0 . . . 185
7.1060 so_33812998_33814842_3 . . . 185
7.1061 so_42369314_42369314_0 . . . 185
7.1062 so_33916113_33916186_7 . . . 185
7.1063 so_23709019_23720793_4 . . . 185
7.1064 so_29111008_29111008_0 . . . 185
7.1065 so_35695071_35695147_1 . . . 185
7.1066 so_18195312_18195408_2 . . . 185
7.1067 so_17045661_17046145_0 . . . 185
7.1068 so_17045661_17046231_0 . . . 185
7.1069 so_33307461_33307583_0 . . . 185
7.1070 so_33307461_33307603_1 . . . 185
7.1071 so_33307461_33307603_0 . . . 185
7.1072 so_38706287_39160364_0 . . . 185
7.1073 so_30852922_30853213_1 . . . 185
7.1074 so_47444081_47444410_1 . . . 185
7.1075 so_47444081_47444410_0 . . . 185
7.1076 so_37631962_37632676_0 . . . 185
7.1077 so_27935026_30153809_3 . . . 185
7.1078 so_37752002_37757636_0 . . . 185
7.1079 so_29035225_29035972_0 . . . 185
7.1080 so_21956017_21961992_0 . . . 185
7.1081 so_32950709_32950795_1 . . . 185
7.1082 so_71416_685239_0 . . . 185
7.1083 so_71416_72599_0 . . . 185
7.1084 so_71416_1932119_0 . . . 185
7.1085 so_11983231_11983423_2 . . . 185
7.1086 so_16549548_16549565_0 . . . 185

7.1087 so_22606392_22606576_0 . . . 185
7.1088 so_23323510_23323551_0 . . . 185
7.1089 so_23323510_23323548_0 . . . 185
7.1090 so_6834581_6834680_1 . . . 185
7.1091 so_28659804_28660475_1 . . . 185
7.1092 so_17888322_17888368_1 . . . 185
7.1093 so_7644113_7644228_0 . . . 185
7.1094 so_27442863_27442974_0 . . . 185
7.1095 so_4851810_4851838_0 . . . 185
7.1096 so_33812998_33814842_1 . . . 185
7.1097 so_29433293_29433534_2 . . . 185
7.1098 so_13992897_13992925_2 . . . 185
7.1099 so_18195312_18195408_9 . . . 185
7.1100 so_33607284_33607345_0 . . . 185
7.1101 so_47444081_47444410_2 . . . 185
7.1102 so_37631962_37632676_2 . . . 185
7.1103 so_27935026_30153809_4 . . . 185
7.1104 so_27935026_30153809_5 . . . 185
7.1105 so_27935026_30153809_7 . . . 185
7.1106 so_45484581_45485116_0 . . . 185
7.1107 so_4167904_4167970_0 . . . 185
7.1108 so_7676817_7676940_3 . . . 185
7.1109 so_7676817_7676940_1 . . . 185
7.1110 so_16736652_16736652_0 . . . 185
7.1111 so_7295632_7295699_1 . . . 185
7.1112 so_31298526_31299313_0 . . . 185
7.1113 so_25528245_25528459_4 . . . 185
7.1114 so_21037316_21038239_1 . . . 185
7.1115 so_39455328_39455421_1 . . . 185
7.1116 so_22606392_22606673_1 . . . 185
7.1117 so_22606392_22606673_2 . . . 185
7.1118 so_1352095_1352111_0 . . . 185
7.1119 so_33812998_33813420_0 . . . 185
7.1120 so_27632176_27632572_2 . . . 185
7.1121 so_27632176_27632176_0 . . . 185
7.1122 so_24566576_24634262_1 . . . 185
7.1123 so_21956017_21956017_0 . . . 185
7.1124 so_34149992_34149992_5 . . . 185
7.1125 so_34149992_34149992_13 . . . 185
7.1126 so_34149992_34149992_18 . . . 185
7.1127 so_30514796_30514941_1 . . . 185
7.1128 so_30514796_30514941_0 . . . 185
7.1129 so_30514796_30514941_2 . . . 185
7.1130 so_13545005_13545065_0 . . . 185
7.1131 so_35536146_35536906_0 . . . 185
7.1132 so_14185352_14185499_0 . . . 185
7.1133 so_14206403_14206403_1 . . . 185
7.1134 so_26744528_26744528_0 . . . 186
7.1135 so_27653959_27653959_0 . . . 186
7.1136 so_6992593_6993152_1 . . . 186
7.1137 so_25831622_25831692_1 . . . 186

7.1138 so_23712025_23712077_0 . . . 186
7.1139 so_20893816_20894013_0 . . . 186
7.1140 so_19795449_19795685_0 . . . 186
7.1141 so_27699714_27699714_1 . . . 186
7.1142 Namespaces . . . 187
7.1143 so_34247975_34249649_4 . . . 187
7.1144 so_13484387_13484574_2 . . . 187
7.1145 so_24220127_24220504_2 . . . 187
7.1146 so_24220127_24220504_1 . . . 187
7.1147 so_32655897_32659227_1 . . . 187
7.1148 so_21286497_21286621_1 . . . 187
7.1149 so_23603576_23603877_1 . . . 187
7.1150 so_16772759_16772837_0 . . . 187
7.1151 so_29775091_29775973_4 . . . 187
7.1152 so_29775091_29775973_1 . . . 187
7.1153 so_13484387_13484574_0 . . . 187
7.1154 so_32595213_32595306_3 . . . 188
7.1155 so_24293311_24293311_1 . . . 188
7.1156 so_34247975_34248633_22 . . . 188
7.1157 so_14144953_14145051_0 . . . 188
7.1158 so_44994461_44994784_0 . . . 188
7.1159 so_24219690_24219690_0 . . . 188
7.1160 so_29261910_29262172_0 . . . 188
7.1161 so_17758659_17758778_1 . . . 188
7.1162 so_17758659_17758778_0 . . . 188
7.1163 so_17758659_17758778_2 . . . 188
7.1164 so_34247975_34248633_23 . . . 188
7.1165 so_24228299_24228299_0 . . . 188
7.1166 so_24228299_24229413_2 . . . 188
7.1167 so_24228299_24229413_1 . . . 188
7.1168 so_27252466_27252466_0 . . . 188
7.1169 so_25435897_25436072_0 . . . 188
7.1170 so_25435897_25436164_0 . . . 188
7.1171 so_24220127_24220504_5 . . . 188
7.1172 so_24220127_24220504_4 . . . 188
7.1173 so_24220127_24220127_0 . . . 188
7.1174 so_24220127_24220504_6 . . . 188
7.1175 so_24220127_24220504_7 . . . 188
7.1176 so_43787462_43792468_0 . . . 188
7.1177 so_34247975_34248633_24 . . . 188
7.1178 so_7181809_7181905_2 . . . 189
7.1179 so_7181809_7181905_1 . . . 189
7.1180 so_7670978_7671418_2 . . . 189
7.1181 so_7670978_7671006_0 . . . 189
7.1182 so_7670978_7671418_1 . . . 189
7.1183 so_7670978_7671006_1 . . . 189
7.1184 so_2094072_2094214_0 . . . 189
7.1185 so_2094072_2094214_1 . . . 189
7.1186 so_31900112_31900313_1 . . . 189
7.1187 so_31900112_31900313_0 . . . 189
7.1188 so_26026273_26026273_1 . . . 189

7.1189 so_26070916_26071134_1 . . . 189
7.1190 so_26070916_26071134_0 . . . 189
7.1191 so_8329871_8329930_2 . . . 189
7.1192 so_8329871_8329930_3 . . . 189
7.1193 so_24409996_24409996_2 . . . 189
7.1194 so_24409996_24410051_0 . . . 189
7.1195 so_24409996_24409996_1 . . . 189
7.1196 so_7436808_7436834_1 . . . 189
7.1197 so_4181059_4181072_0 . . . 189
7.1198 so_20208591_20208591_1 . . . 189
7.1199 so_20208591_20208591_0 . . . 189
7.1200 so_3673353_3673368_0 . . . 189
7.1201 so_3673353_3673368_1 . . . 189
7.1202 so_35288838_35288838_1 . . . 189
7.1203 so_35288838_35289169_2 . . . 189
7.1204 so_35288838_35290352_2 . . . 189
7.1205 so_13078893_13079051_0 . . . 189
7.1206 so_13078893_13079051_1 . . . 189
7.1207 so_36890298_36900118_2 . . . 189
7.1208 so_36890298_36900118_1 . . . 189
7.1209 so_9622874_9622896_1 . . . 189
7.1210 so_8775563_8775617_3 . . . 189
7.1211 so_8775563_8775617_1 . . . 189
7.1212 so_13622781_13622781_0 . . . 189
7.1213 so_8001047_8001177_2 . . . 189
7.1214 so_2378150_2378207_0 . . . 189
7.1215 so_2378150_2378207_1 . . . 189
7.1216 so_25624381_25624381_0 . . . 189
7.1217 so_31729985_31731551_1 . . . 189
7.1218 so_32074340_32074528_3 . . . 189
7.1219 so_25566589_25566724_0 . . . 189
7.1220 so_37897810_37897810_0 . . . 189
7.1221 so_29928957_29929143_1 . . . 189
7.1222 so_17611000_17611426_0 . . . 189
7.1223 so_16370599_16371086_0 . . . 189
7.1224 so_31729985_31731551_2 . . . 189
7.1225 so_9608798_9608888_0 . . . 190
7.1226 so_37328444_37328923_0 . . . 190
7.1227 so_23451463_23451463_0 . . . 190
7.1228 so_623903_623937_1 . . . 190
7.1229 so_8207633_8284809_0 . . . 190
7.1230 so_21602957_21605768_2 . . . 190
7.1231 so_4757871_4757918_2 . . . 190
7.1232 so_15698432_15698609_0 . . . 190
7.1233 so_30418270_30418323_1 . . . 190
7.1234 so_30418270_30418323_0 . . . 190
7.1235 so_30418270_30418339_0 . . . 190
7.1236 so_44186063_44187130_1 . . . 190
7.1237 so_41769510_41769510_0 . . . 190
7.1238 so_41769510_41769636_0 . . . 190
7.1239 so_8393371_8393371_0 . . . 190

7.1240 so_10937096_10937195_1 . . . 190
7.1241 so_47136153_47136624_1 . . . 190
7.1242 so_47136153_47136287_0 . . . 190
7.1243 so_24600050_24600424_0 . . . 190
7.1244 so_7785886_7785935_0 . . . 190
7.1245 so_41182812_41184270_0 . . . 190
7.1246 so_6003373_6003411_0 . . . 190
7.1247 so_23082525_23082900_1 . . . 190
7.1248 so_23082525_23082525_1 . . . 190
7.1249 so_23082525_23082525_0 . . . 190
7.1250 so_32815210_32815366_0 . . . 190
7.1251 so_12573816_35891188_0 . . . 190
7.1252 so_21503621_21503621_0 . . . 190
7.1253 so_6321191_6321191_1 . . . 190
7.1254 so_33283092_33283279_0 . . . 190
7.1255 so_27974521_27974902_1 . . . 190
7.1256 so_27974521_27974902_2 . . . 190
7.1257 so_3593032_3593047_0 . . . 190
7.1258 so_15770451_15770612_0 . . . 190
7.1259 so_4274386_4279182_0 . . . 190
7.1260 so_4274386_4279182_1 . . . 190
7.1261 so_13898053_13898053_0 . . . 190
7.1262 so_13898053_13898724_0 . . . 190
7.1263 so_23540764_23541148_0 . . . 190
7.1264 so_36025007_36025007_0 . . . 190
7.1265 so_31005610_31008701_0 . . . 190
7.1266 so_31005610_31005610_0 . . . 190
7.1267 so_35463169_35482857_0 . . . 190
7.1268 so_28708829_28708829_0 . . . 190
7.1269 so_24423481_24424033_1 . . . 190
7.1270 so_8514686_8514899_0 . . . 190
7.1271 so_24167949_24168770_0 . . . 190
7.1272 so_23704266_23704331_1 . . . 190
7.1273 so_8752587_8753091_0 . . . 190
7.1274 so_16370599_16371086_1 . . . 190
7.1275 so_27917680_27922878_3 . . . 190
7.1276 so_29574616_29574616_0 . . . 190
7.1277 so_4210828_4211151_0 . . . 190
7.1278 so_28006660_28006861_0 . . . 190
7.1279 so_4492062_25157623_0 . . . 190
7.1280 so_4492062_4492524_0 . . . 190
7.1281 so_25672745_25672745_0 . . . 190
7.1282 so_27418384_27418628_0 . . . 190
7.1283 so_31348475_31348673_0 . . . 190
7.1284 so_31348475_31348965_0 . . . 190
7.1285 so_31348475_31348475_0 . . . 190
7.1286 so_13938037_13938383_0 . . . 190
7.1287 so_21029511_21029511_0 . . . 190
7.1288 so_45907561_45907561_0 . . . 190
7.1289 so_7809468_7809637_0 . . . 190
7.1290 so_8619356_8620289_0 . . . 190

7.1291 so_1634589_1634589_0 . . . 190
7.1292 so_28241874_28242034_2 . . . 190
7.1293 so_28241874_28242038_3 . . . 190
7.1294 so_45964105_45964105_0 . . . 190
7.1295 so_46073787_46073787_4 . . . 190
7.1296 so_46073787_46075199_1 . . . 190
7.1297 so_46073787_46073787_1 . . . 190
7.1298 so_46073787_46075199_0 . . . 190
7.1299 so_25692402_25692496_2 . . . 190
7.1300 so_31772588_31775526_1 . . . 190
7.1301 so_31772588_31773372_1 . . . 190
7.1302 so_8913298_8913361_0 . . . 190
7.1303 so_18757641_18758653_0 . . . 190
7.1304 so_31004483_31004839_0 . . . 190
7.1305 so_31004483_31004839_1 . . . 190
7.1306 so_28003493_28003780_0 . . . 190
7.1307 so_26058966_26059248_0 . . . 191
7.1308 so_5510576_5510808_0 . . . 191
7.1309 so_16772759_16772837_2 . . . 191
7.1310 so_47006008_47006105_1 . . . 191
7.1311 so_29775091_29775091_0 . . . 191
7.1312 so_31707358_31707753_0 . . . 191
7.1313 so_31707358_31707753_3 . . . 191
7.1314 so_438192_438214_0 . . . 191
7.1315 so_31711183_31711183_1 . . . 191
7.1316 so_31711183_31711183_3 . . . 191
7.1317 so_32265938_32266959_1 . . . 191
7.1318 so_27158750_27158750_0 . . . 191
7.1319 so_27158750_27159170_1 . . . 191
7.1320 so_27402292_27404546_9 . . . 191
7.1321 so_31518214_31518990_0 . . . 191
7.1322 so_31518214_31518990_1 . . . 191
7.1323 so_3934862_3934947_0 . . . 191
7.1324 so_31708705_31710965_3 . . . 191
7.1325 so_31708705_31708705_3 . . . 191
7.1326 so_31708705_31708705_0 . . . 191
7.1327 so_6942949_6946317_0 . . . 191
7.1328 so_15124077_15124077_1 . . . 191
7.1329 so_20548370_20631809_1 . . . 191
7.1330 so_20548370_20631809_0 . . . 191
7.1331 so_2166271_2166366_1 . . . 191
7.1332 so_24366565_24367452_2 . . . 191
7.1333 so_43787462_43792468_6 . . . 191
7.1334 so_42388431_42573292_1 . . . 191
7.1335 so_42388431_42573292_4 . . . 191
7.1336 so_27158750_27158750_1 . . . 191
7.1337 so_25547514_26895652_0 . . . 191
7.1338 so_12074058_12074308_0 . . . 191
7.1339 so_15079947_15080960_0 . . . 192
7.1340 so_25863485_25865297_3 . . . 192
7.1341 so_27158750_27158750_2 . . . 192

7.1342 so_19399644_19405833_0 . . . 192
7.1343 so_31396895_31397049_0 . . . 192
7.1344 so_25653647_25653843_0 . . . 192
7.1345 so_3546340_3546582_1 . . . 192
7.1346 so_39702532_39702920_0 . . . 192
7.1347 so_6275383_6275532_0 . . . 192
7.1348 so_38608687_38609839_1 . . . 192
7.1349 so_47556287_47556411_1 . . . 192
7.1350 so_34411588_34412317_0 . . . 192
7.1351 so_24212921_24213261_1 . . . 193
7.1352 so_24212921_24213261_2 . . . 193
7.1353 so_15450029_15450111_0 . . . 193
7.1354 so_32794789_32795072_1 . . . 193
7.1355 so_32794789_32795072_2 . . . 193
7.1356 so_438192_438717_0 . . . 193
7.1357 so_41033384_41034562_1 . . . 193
7.1358 so_5100451_5100719_0 . . . 193
7.1359 so_27348968_27348985_0 . . . 193
7.1360 so_33212696_33212696_0 . . . 193
7.1361 so_23847774_23851373_0 . . . 193
7.1362 so_31343142_31348802_1 . . . 193
7.1363 so_31343142_31343676_1 . . . 193
7.1364 so_31224886_31234579_0 . . . 193
7.1365 so_31224886_31225016_0 . . . 193
7.1366 so_4252451_4252451_0 . . . 193
7.1367 so_4252451_4252483_0 . . . 193
7.1368 so_27977700_27977700_0 . . . 193
7.1369 so_31343142_31348802_2 . . . 194
7.1370 so_31711183_31711422_0 . . . 194
7.1371 so_27316607_27316980_0 . . . 194
7.1372 so_3544824_3544838_1 . . . 194
7.1373 so_15079947_15080960_1 . . . 195
7.1374 so_5617043_5617327_2 . . . 195
7.1375 so_15148087_15148532_0 . . . 195
7.1376 so_15148087_15148532_1 . . . 195
7.1377 so_45019001_45019001_0 . . . 195
7.1378 so_7645394_7646071_1 . . . 195
7.1379 so_31343142_31343676_0 . . . 195
7.1380 so_31343142_31348802_5 . . . 195
7.1381 so_6368005_6368158_0 . . . 195
7.1382 so_34158902_34159684_1 . . . 195
7.1383 so_33694192_33696684_5 . . . 195
7.1384 so_33694192_33696684_1 . . . 195
7.1385 so_33694192_33696684_3 . . . 195
7.1386 so_5617043_5617327_1 . . . 195
7.1387 so_17602149_17602451_0 . . . 195
7.1388 so_8549362_8549519_1 . . . 195
7.1389 so_8549362_8549519_0 . . . 195
7.1390 so_15148087_15148532_3 . . . 195
7.1391 so_46805276_46805276_0 . . . 195
7.1392 so_36635372_36635702_0 . . . 195

7.1393 so_3544824_3544838_0 . . . 195
7.1394 so_24409307_24409775_0 . . . 195
7.1395 so_46769112_46769854_0 . . . 195
7.1396 so_29317650_29318289_0 . . . 195
7.1397 so_7645394_7646071_2 . . . 195
7.1398 so_36409348_36411098_0 . . . 195
7.1399 so_15079947_15080960_2 . . . 196
7.1400 so_31343142_31348802_6 . . . 196
7.1401 so_23148158_23148648_0 . . . 196
7.1402 so_33721762_33730436_1 . . . 196
7.1403 so_34323791_34323857_0 . . . 196
7.1404 so_31518214_31518760_1 . . . 196
7.1405 so_46750163_46750570_0 . . . 196
7.1406 so_6942949_6946317_2 . . . 196
7.1407 so_20548370_20631809_2 . . . 196
7.1408 so_806561_806644_0 . . . 196
7.1409 so_4094534_4094751_1 . . . 196
7.1410 so_30036255_30036548_0 . . . 196
7.1411 so_28223930_28224007_0 . . . 196
7.1412 so_28548789_28549139_0 . . . 197
7.1413 so_2181350_2181396_0 . . . 197
7.1414 so_27158750_27158750_3 . . . 197
7.1415 so_1071119_1071461_2 . . . 197
7.1416 so_18287151_18287422_3 . . . 197
7.1417 so_5100451_5100719_2 . . . 197
7.1418 so_6326805_6326856_0 . . . 197
7.1419 so_3934862_3934947_1 . . . 197
7.1420 so_33887839_33888228_0 . . . 197
7.1421 so_27903704_27903879_0 . . . 197
7.1422 so_27433544_27433544_0 . . . 197
7.1423 so_18556714_18556784_0 . . . 197
7.1424 so_32265938_32266959_0 . . . 197
7.1425 so_2315885_2315890_0 . . . 197
7.1426 so_27544893_27545083_1 . . . 197
7.1427 so_24319602_24319682_0 . . . 197
7.1428 so_10741428_10741741_0 . . . 197
7.1429 so_10741428_10741428_0 . . . 197
7.1430 so_3708626_3709123_0 . . . 197
7.1431 so_39799873_39833904_1 . . . 197
7.1432 so_30032332_30033550_0 . . . 197
7.1433 so_30032332_30033553_0 . . . 197
7.1434 so_44894400_44894477_0 . . . 197
7.1435 so_46168188_46168323_0 . . . 197
7.1436 so_16081289_16083098_0 . . . 197
7.1437 so_40489677_40490475_0 . . . 197
7.1438 so_23582526_23583080_0 . . . 197
7.1439 so_20853784_20854689_0 . . . 197
7.1440 so_44994461_44994461_0 . . . 197
7.1441 so_44912330_44912330_0 . . . 197
7.1442 so_16093226_16093311_0 . . . 197
7.1443 so_31494906_31495054_1 . . . 197

7.1444 so_31494906_31494906_0 . . . 197
7.1445 so_32965707_32966693_0 . . . 197
7.1446 so_40164871_40165003_0 . . . 197
7.1447 so_12516476_12516476_0 . . . 197
7.1448 so_18933857_18934399_1 . . . 197
7.1449 so_31702956_31702956_2 . . . 197
7.1450 so_31702956_31703356_1 . . . 197
7.1451 so_27903704_27903879_1 . . . 197
7.1452 so_27433544_27434098_1 . . . 197
7.1453 so_27544893_27545083_2 . . . 197
7.1454 so_10741428_10741741_1 . . . 197
7.1455 so_39799873_39833904_2 . . . 197
7.1456 so_16081289_16081535_0 . . . 197
7.1457 so_20853784_20854689_1 . . . 197
7.1458 so_20853784_20853784_0 . . . 197
7.1459 so_13078893_13079051_2 . . . 198
7.1460 so_8775563_8775617_0 . . . 198
7.1461 so_31707358_31707358_0 . . . 199
7.1462 so_36204506_36204704_0 . . . 199
7.1463 so_36204506_36204674_0 . . . 199
7.1464 so_31711183_31711699_0 . . . 199
7.1465 so_31711183_31711183_0 . . . 199
7.1466 so_29883977_29883978_0 . . . 199
7.1467 so_36644566_36645467_0 . . . 199
7.1468 so_33313853_33318306_0 . . . 199
7.1469 so_16081289_16081535_1 . . . 199
7.1470 so_37301946_37303140_6 . . . 199
7.1471 so_31702956_31703369_3 . . . 199
7.1472 so_31702956_31703356_0 . . . 199
7.1473 The `asm` declaration . . . 200
7.1474 so_13870489_13870521_5 . . . 200
7.1475 so_13870489_13870707_2 . . . 200
7.1476 so_27432621_28783749_0 . . . 200
7.1477 Linkage specifications . . . 200
7.1478 so_21955288_21955322_1 . . . 200
7.1479 so_17117861_17118037_1 . . . 200
7.1480 so_45265690_45372988_0 . . . 200
7.1481 so_37948209_37948575_0 . . . 200
7.1482 so_15536488_15536838_0 . . . 200
7.1483 so_2068022_2068048_3 . . . 200
7.1484 so_12868383_12868383_0 . . . 200
7.1485 so_8948443_8949164_1 . . . 200
7.1486 so_12474351_12474351_0 . . . 200
7.1487 so_26160962_26161361_0 . . . 200
7.1488 so_14395192_14396177_0 . . . 200
7.1489 so_21484500_21484617_1 . . . 200
7.1490 so_21484500_21484617_0 . . . 200
7.1491 so_21026264_21026358_1 . . . 200
7.1492 so_21026264_21026358_2 . . . 200
7.1493 so_14616979_14617173_0 . . . 200
7.1494 so_21421720_21421720_0 . . . 201

7.1495 so_4186165_4186424_0 . . . 201
7.1496 so_4877705_4877705_0 . . . 201
7.1497 so_4866433_4866508_0 . . . 201
7.1498 so_45265690_45265690_0 . . . 201
7.1499 so_27441445_27441622_0 . . . 201
7.1500 so_27441445_27441622_1 . . . 201
7.1501 so_18028342_18028417_0 . . . 201
7.1502 so_28129777_28133425_0 . . . 201
7.1503 so_28129777_28133425_1 . . . 201
7.1504 so_37088979_37095588_0 . . . 201
7.1505 so_1025345_1950150_1 . . . 201
7.1506 so_11873939_11874337_0 . . . 201
7.1507 so_21000882_21004439_1 . . . 201
7.1508 so_6932587_6932634_0 . . . 201
7.1509 so_31729985_31731551_3 . . . 201
7.1510 so_5589381_5589381_0 . . . 201
7.1511 so_26160962_26161361_1 . . . 201
7.1512 so_14395192_14396177_1 . . . 201
7.1513 so_45358729_45359202_0 . . . 201
7.1514 so_17195773_17196371_2 . . . 201
7.1515 so_1906000_1906168_1 . . . 202
7.1516 so_24888001_24888067_1 . . . 202
7.1517 so_37927388_37927669_0 . . . 202
7.1518 so_26160465_26160651_2 . . . 202
7.1519 so_29463565_29463646_0 . . . 202
7.1520 so_26160962_26160962_4 . . . 202
7.1521 so_45122265_45122265_2 . . . 202
7.1522 so_31707358_31707646_0 . . . 202
7.1523 so_23543453_23544915_0 . . . 202
7.1524 so_12355161_12355267_1 . . . 202
7.1525 so_27777071_32970511_1 . . . 202
7.1526 so_32851184_32851722_0 . . . 202
7.1527 so_27299158_27305232_1 . . . 202
7.1528 so_27299158_27305232_0 . . . 202
7.1529 so_24888001_24888067_0 . . . 202
7.1530 so_27683995_27683995_0 . . . 202
7.1531 so_31005610_31008701_2 . . . 202
7.1532 so_23991085_23991160_0 . . . 202
7.1533 so_4866043_4866117_1 . . . 202
7.1534 so_4866043_4866087_0 . . . 202
7.1535 so_8986498_8999649_1 . . . 202
7.1536 so_28484557_28484773_0 . . . 202
7.1537 so_27675547_27675672_1 . . . 202
7.1538 so_28763052_28763052_5 . . . 202
7.1539 so_7626774_7626830_2 . . . 203
7.1540 so_21026264_21026358_3 . . . 203
7.1541 so_23891058_23891697_1 . . . 203
7.1542 so_1906000_1906168_0 . . . 203
7.1543 so_31882729_31882963_0 . . . 203
7.1544 so_31882729_31882963_1 . . . 203
7.1545 so_8948443_8949164_2 . . . 203

7.1546 Attributes . . . 203
7.1547 so_7414829_7414829_0 . . . 204
7.1548 so_36811786_36811858_1 . . . 204
7.1549 so_15788947_34006668_1 . . . 204
7.1550 so_7414829_7416898_1 . . . 204
7.1551 so_47104799_47105294_1 . . . 204
7.1552 so_7414829_7414829_1 . . . 204
7.1553 so_36811786_36811858_3 . . . 204
7.1554 so_27468666_27468687_0 . . . 205
7.1555 so_39411971_39412209_0 . . . 205
7.1556 so_8276885_8276885_0 . . . 205
7.1557 so_30734351_30734351_0 . . . 205
7.1558 so_15788947_15788947_0 . . . 205
7.1559 so_15788947_15791742_1 . . . 205
7.1560 so_15788947_15912208_2 . . . 205
7.1561 so_15788947_34006668_0 . . . 205
7.1562 so_7501211_7501423_1 . . . 205
7.1563 so_15523537_15527364_2 . . . 205
7.1564 so_7501211_7501423_2 . . . 205
7.1565 so_34376921_34377119_0 . . . 205
7.1566 so_34376921_34377119_2 . . . 205
7.1567 so_24045850_24046184_4 . . . 205
7.1568 so_43077548_43078625_0 . . . 205
7.1569 so_18978006_18978080_0 . . . 205
7.1570 so_26888805_26889285_0 . . . 206
7.1571 so_26888805_26889159_0 . . . 206
7.1572 so_38623143_38623312_0 . . . 206
7.1573 so_38623143_38623312_1 . . . 206
7.1574 so_38623143_38623322_0 . . . 206
7.1575 so_11314580_11314580_0 . . . 207
7.1576 so_26888805_26889285_1 . . . 207
7.1577 so_295120_21192071_0 . . . 208
7.1578 so_46688604_46743721_0 . . . 208
7.1579 so_22134414_22137806_0 . . . 208
7.1580 so_22134414_22137806_2 . . . 208
7.1581 so_11017512_21190574_0 . . . 208
7.1582 so_41760241_41762123_0 . . . 208
7.1583 so_46688604_46743721_1 . . . 208
7.1584 so_22134414_22137806_3 . . . 208
7.1585 so_38513812_38516575_0 . . . 208
7.1586 so_22134414_22137806_4 . . . 208
7.1587 so_22134414_22137806_5 . . . 208

8 Declarators **209**
8.1 so_34247975_34249649_1 . . . 209
8.2 so_35207678_35211117_1 . . . 209
8.3 so_16821956_16821988_0 . . . 209
8.4 so_6337426_6337447_0 . . . 209
8.5 so_15248603_15248697_0 . . . 209
8.6 so_31535401_31535603_1 . . . 209
8.7 so_29761114_29761138_3 . . . 209

8.8 so_29761114_29761138_1 . . . 209
8.9 so_29761114_29761138_0 . . . 209
8.10 so_24224115_24225154_1 . . . 209
8.11 so_24224115_24225154_0 . . . 209
8.12 so_12729962_12729991_1 . . . 209
8.13 so_12729962_12729991_0 . . . 209
8.14 so_12729962_12729991_5 . . . 209
8.15 so_12729962_12729991_6 . . . 209
8.16 so_27432506_27432506_15 . . . 209
8.17 so_27432506_27432506_14 . . . 209
8.18 so_6414030_6414247_1 . . . 209
8.19 so_6414030_6414247_0 . . . 209
8.20 so_16642294_16642294_0 . . . 209
8.21 so_35561075_35561262_0 . . . 209
8.22 so_35561075_35561262_1 . . . 209
8.23 so_35561075_35564836_0 . . . 209
8.24 so_35561075_35561242_0 . . . 209
8.25 so_35561075_35561242_1 . . . 209
8.26 so_10347298_10347528_0 . . . 209
8.27 so_24836062_24836148_4 . . . 209
8.28 so_21577466_21577912_1 . . . 209
8.29 so_21577466_21577912_0 . . . 209
8.30 so_21577466_21577912_2 . . . 209
8.31 so_29479692_29479692_0 . . . 210
8.32 Type names . . . 210
8.33 so_23015482_23015727_1 . . . 210
8.34 so_28658451_28659194_0 . . . 210
8.35 so_36247823_36247823_0 . . . 210
8.36 so_35848012_35848410_0 . . . 210
8.37 so_16498969_16504831_0 . . . 210
8.38 Ambiguity resolution . . . 211
8.39 so_5297763_5300674_1 . . . 211
8.40 so_17994955_18000366_2 . . . 211
8.41 so_1652396_8461052_0 . . . 211
8.42 so_26399686_26399786_1 . . . 211
8.43 so_24116817_24116818_5 . . . 211
8.44 so_7007817_7009105_1 . . . 211
8.45 so_47190568_47191759_13 . . . 211
8.46 so_47190568_47191759_12 . . . 211
8.47 so_47190568_47191759_10 . . . 211
8.48 so_47190568_47191759_4 . . . 211
8.49 so_38951362_38951441_2 . . . 211
8.50 so_33301356_33301356_0 . . . 211
8.51 so_14403946_14404900_1 . . . 211
8.52 so_39281235_39281264_0 . . . 211
8.53 so_7007817_7009105_2 . . . 211
8.54 so_26282474_26282755_0 . . . 211
8.55 so_7007817_7009105_3 . . . 212
8.56 Meaning of declarators . . . 213
8.57 so_31496347_31496793_0 . . . 213
8.58 so_15788947_15912208_1 . . . 213

8.59 so_25354833_25355365_2 . . . 213
8.60 so_17958497_17958497_1 . . . 213
8.61 so_17958497_17958497_0 . . . 213
8.62 so_23015482_23016270_0 . . . 213
8.63 so_34318661_34319363_0 . . . 213
8.64 so_41769510_41769636_1 . . . 213
8.65 so_37897810_37898353_2 . . . 213
8.66 so_18398409_18400950_1 . . . 213
8.67 so_4417219_4417421_0 . . . 213
8.68 so_27081344_27081344_2 . . . 213
8.69 so_27081344_27081344_3 . . . 213
8.70 so_31729985_31735140_0 . . . 213
8.71 so_33642510_33642510_0 . . . 213
8.72 so_15692660_15692739_0 . . . 213
8.73 so_36113229_36113490_0 . . . 213
8.74 so_38276005_38280119_0 . . . 213
8.75 so_15424831_15425371_0 . . . 213
8.76 so_23468208_23471815_0 . . . 213
8.77 so_21763935_21764310_5 . . . 213
8.78 so_21763935_21764310_4 . . . 213
8.79 so_21763935_21764310_3 . . . 213
8.80 so_21763935_21764310_2 . . . 213
8.81 so_34417071_34417330_3 . . . 213
8.82 so_34417071_34417330_5 . . . 213
8.83 so_28283215_28283405_5 . . . 213
8.84 so_28283215_28283405_6 . . . 213
8.85 so_45991094_45991162_0 . . . 213
8.86 so_45991094_45991162_1 . . . 213
8.87 so_24623071_24623317_2 . . . 213
8.88 so_24623071_24623317_4 . . . 213
8.89 so_23895545_23895705_2 . . . 213
8.90 so_23895545_23895705_5 . . . 213
8.91 so_23895545_23895705_6 . . . 213
8.92 so_23895545_23895705_3 . . . 213
8.93 so_27139006_27139046_3 . . . 213
8.94 so_27139006_27139046_2 . . . 213
8.95 so_23468208_23471815_1 . . . 214
8.96 so_24260456_24261674_2 . . . 214
8.97 so_13296706_13297464_0 . . . 214
8.98 so_6071253_6071291_0 . . . 214
8.99 so_27256516_27256903_0 . . . 214
8.100 so_29109500_29109711_0 . . . 214
8.101 so_13779273_13779273_0 . . . 214
8.102 so_40568170_40568833_1 . . . 214
8.103 so_40568170_40568833_2 . . . 214
8.104 so_38044834_38044974_2 . . . 214
8.105 so_20443992_20444093_0 . . . 214
8.106 so_20443992_20444093_1 . . . 214
8.107 so_3553386_3553535_0 . . . 214
8.108 so_31510673_31511079_1 . . . 214
8.109 so_31510673_31511079_0 . . . 214

8.110 so_12660845_12660897_0 . . . 214
8.111 so_28359555_28363653_0 . . . 214
8.112 so_20543146_20543284_3 . . . 214
8.113 so_36730652_36730709_0 . . . 214
8.114 so_11739804_11739951_1 . . . 214
8.115 so_37202716_37202863_1 . . . 214
8.116 so_24827592_24828578_1 . . . 214
8.117 so_27080061_27080417_2 . . . 214
8.118 so_19262062_19262081_0 . . . 214
8.119 so_2627166_3598165_0 . . . 214
8.120 so_457896_457896_0 . . . 214
8.121 so_29102629_29102749_0 . . . 214
8.122 so_31262942_31283235_0 . . . 214
8.123 so_3801982_3802028_0 . . . 214
8.124 so_26083067_26083067_0 . . . 214
8.125 so_35352168_35352325_0 . . . 214
8.126 so_39631317_39631367_0 . . . 214
8.127 so_27727637_27728034_0 . . . 214
8.128 so_29540907_29541200_0 . . . 214
8.129 so_2220230_2220254_5 . . . 215
8.130 so_31510673_31511079_2 . . . 215
8.131 so_37640379_37640407_0 . . . 216
8.132 so_26631169_26631169_1 . . . 216
8.133 so_3931167_3931551_2 . . . 216
8.134 so_35574143_35574270_0 . . . 216
8.135 so_15229055_15229287_1 . . . 216
8.136 so_14799690_14799690_0 . . . 216
8.137 so_20016049_20016098_0 . . . 216
8.138 so_20993147_20993504_0 . . . 216
8.139 so_22459167_22459395_1 . . . 216
8.140 so_23959119_23959216_0 . . . 216
8.141 so_32920830_32920949_0 . . . 216
8.142 so_35087542_35087542_1 . . . 216
8.143 so_31521116_31521236_0 . . . 216
8.144 so_21078243_21078418_0 . . . 216
8.145 so_29322688_29322713_0 . . . 216
8.146 so_17853767_17854139_1 . . . 216
8.147 so_28486257_28487370_0 . . . 216
8.148 so_15972474_15972579_1 . . . 216
8.149 so_30859709_30859758_0 . . . 216
8.150 so_15486052_15486075_0 . . . 216
8.151 so_5719636_5730378_4 . . . 216
8.152 so_24674609_24674663_0 . . . 216
8.153 so_37719944_37720099_0 . . . 216
8.154 so_22116997_22117225_1 . . . 216
8.155 so_9849041_9849228_0 . . . 216
8.156 so_13964690_13964709_1 . . . 216
8.157 so_35500250_35500308_0 . . . 216
8.158 so_22098834_22098970_10 . . . 216
8.159 so_19554198_19555419_3 . . . 216
8.160 so_34588945_34589189_2 . . . 216

8.161 so_27802605_27802667_0 . . . 216
8.162 so_27802605_27802667_1 . . . 216
8.163 so_25544460_25544511_0 . . . 216
8.164 so_21224407_21224546_0 . . . 216
8.165 so_17142130_17147477_2 . . . 216
8.166 so_1155142_1155148_1 . . . 216
8.167 so_7622115_7622486_0 . . . 216
8.168 so_13779273_13779331_0 . . . 216
8.169 so_4285895_4285991_0 . . . 216
8.170 so_35069224_35069579_2 . . . 216
8.171 so_6433339_6434112_0 . . . 216
8.172 so_24396987_24397182_0 . . . 216
8.173 so_2934021_2934291_0 . . . 216
8.174 so_31240981_31240981_0 . . . 216
8.175 so_28573215_28573396_0 . . . 216
8.176 so_17167133_17167167_0 . . . 216
8.177 so_25674834_25674834_2 . . . 216
8.178 so_4364536_4364586_0 . . . 216
8.179 so_2894891_2894949_0 . . . 216
8.180 so_2027122_2027235_0 . . . 216
8.181 so_28359555_28363653_2 . . . 216
8.182 so_4632528_4632601_0 . . . 216
8.183 so_26290598_26291984_0 . . . 216
8.184 so_47237933_47239853_0 . . . 216
8.185 so_4190585_4190689_1 . . . 216
8.186 so_19482671_19482858_3 . . . 216
8.187 so_37724668_37725173_1 . . . 216
8.188 so_4891146_4891191_0 . . . 216
8.189 so_17106448_17107043_5 . . . 216
8.190 so_11748652_11748759_0 . . . 216
8.191 so_8804612_8808865_0 . . . 216
8.192 so_6246365_6258431_3 . . . 216
8.193 so_23482688_23482915_0 . . . 216
8.194 so_11999037_11999097_0 . . . 216
8.195 so_40461797_40461927_0 . . . 216
8.196 so_24794740_24794740_0 . . . 216
8.197 so_24794740_24795077_2 . . . 216
8.198 so_24794740_24795077_1 . . . 216
8.199 so_21232145_21232381_0 . . . 216
8.200 so_5838895_5839026_1 . . . 216
8.201 so_657964_657968_0 . . . 216
8.202 so_1179937_1179951_1 . . . 216
8.203 so_3858470_3858485_0 . . . 216
8.204 so_25844637_25844808_3 . . . 216
8.205 so_9055778_9055797_0 . . . 216
8.206 so_15479570_15479643_0 . . . 216
8.207 so_12979625_12980934_0 . . . 216
8.208 so_25381126_25381336_0 . . . 216
8.209 so_5460562_5460562_0 . . . 216
8.210 so_8336274_8336392_2 . . . 216
8.211 so_2321578_2321591_0 . . . 216

8.212 so_33790065_33791273_0 . . . 216
8.213 so_1164266_1164306_0 . . . 216
8.214 so_39929099_39929147_0 . . . 216
8.215 so_22098834_22098970_9 . . . 216
8.216 so_22098834_22098834_1 . . . 216
8.217 so_22098834_22098970_11 . . . 216
8.218 so_17121822_17122450_0 . . . 216
8.219 so_19554198_19555419_1 . . . 216
8.220 so_2727834_2727872_0 . . . 216
8.221 so_27256516_27256623_0 . . . 216
8.222 so_37700854_37700945_0 . . . 216
8.223 so_24549008_24550557_1 . . . 216
8.224 so_19107851_19108471_0 . . . 216
8.225 so_20543146_20543284_0 . . . 216
8.226 so_11739804_11739951_2 . . . 216
8.227 so_5108111_5108295_0 . . . 216
8.228 so_11999037_11999097_1 . . . 216
8.229 so_24621954_24622052_0 . . . 216
8.230 so_27365839_27365910_1 . . . 216
8.231 so_13032670_13032765_0 . . . 216
8.232 so_27162225_27162410_1 . . . 216
8.233 so_24978819_24978892_0 . . . 216
8.234 so_27766601_27766601_0 . . . 216
8.235 so_29540907_29541200_1 . . . 216
8.236 so_23142361_23142429_0 . . . 216
8.237 so_31206462_31206733_2 . . . 216
8.238 so_30139580_30141622_3 . . . 216
8.239 so_21409358_21409866_2 . . . 216
8.240 so_26200491_26200699_1 . . . 216
8.241 so_17142130_17147477_3 . . . 217
8.242 so_14910616_14910723_1 . . . 217
8.243 so_8026483_8026500_0 . . . 217
8.244 so_8026483_8026483_0 . . . 217
8.245 so_21952386_21952738_0 . . . 217
8.246 so_40147385_40147385_0 . . . 217
8.247 so_7245404_7245602_0 . . . 217
8.248 so_6662917_6662986_1 . . . 217
8.249 so_8336274_8336392_0 . . . 217
8.250 so_8508266_8508299_0 . . . 217
8.251 so_39946420_39946420_0 . . . 217
8.252 so_40590216_40590301_0 . . . 217
8.253 so_27632808_27633033_2 . . . 217
8.254 so_2717671_2717726_0 . . . 217
8.255 so_27244588_27245211_1 . . . 217
8.256 so_3786632_3786632_0 . . . 217
8.257 so_11144579_11144824_0 . . . 217
8.258 so_32090595_32090711_1 . . . 217
8.259 so_22624730_22624774_4 . . . 217
8.260 so_22624730_22624774_2 . . . 217
8.261 so_22624730_22624774_0 . . . 217
8.262 so_24706911_24706911_0 . . . 217

8.263 so_33647319_33647319_1 . . . 217
8.264 so_33647319_33647319_0 . . . 217
8.265 so_28359555_28363653_1 . . . 217
8.266 so_8111357_8111377_1 . . . 217
8.267 so_17804547_17805562_2 . . . 217
8.268 so_17804547_17805562_1 . . . 217
8.269 so_15501470_15504781_3 . . . 217
8.270 so_27117169_27117169_10 . . . 217
8.271 so_27117169_27117169_8 . . . 217
8.272 so_27117169_27117169_4 . . . 217
8.273 so_27117169_27122565_1 . . . 217
8.274 so_27117169_27117272_0 . . . 217
8.275 so_31503575_31503709_0 . . . 217
8.276 so_24151399_24151550_0 . . . 217
8.277 so_46738487_46743940_4 . . . 217
8.278 so_472530_472530_2 . . . 217
8.279 so_34437082_34437307_2 . . . 217
8.280 so_12138243_12175516_0 . . . 217
8.281 so_12138243_12181316_1 . . . 217
8.282 so_34972172_34973290_3 . . . 217
8.283 so_11543625_11544332_1 . . . 217
8.284 so_31643245_31643450_0 . . . 217
8.285 so_31082389_31082764_1 . . . 217
8.286 so_31860300_31860540_0 . . . 217
8.287 so_39878237_39878591_0 . . . 217
8.288 so_39878237_40018150_0 . . . 217
8.289 so_23635310_23635362_2 . . . 217
8.290 so_25660234_25660498_0 . . . 217
8.291 so_10925459_10933049_0 . . . 217
8.292 so_47352663_47352963_0 . . . 217
8.293 so_9573748_9573748_1 . . . 217
8.294 so_4637774_4639568_2 . . . 217
8.295 so_4637774_4638285_0 . . . 217
8.296 so_28119793_28119888_2 . . . 217
8.297 so_28926412_28926516_0 . . . 217
8.298 so_19775954_19775988_3 . . . 217
8.299 so_12879947_12880160_0 . . . 217
8.300 so_29479692_29480560_1 . . . 217
8.301 so_42100976_42212056_0 . . . 217
8.302 so_10353240_10353341_0 . . . 217
8.303 so_43397276_43397453_3 . . . 217
8.304 so_43397276_43397276_0 . . . 217
8.305 so_31120621_31127551_3 . . . 217
8.306 so_9781723_9781886_3 . . . 217
8.307 so_9781723_9781886_2 . . . 217
8.308 so_24826139_24826298_0 . . . 217
8.309 so_46448673_46452913_0 . . . 217
8.310 so_6180012_6180200_0 . . . 217
8.311 so_19756374_19756393_0 . . . 217
8.312 so_21879691_21879799_2 . . . 217
8.313 so_29734554_29734680_0 . . . 217

8.314 so_45902923_45903381_1 . . . 217
8.315 so_38057210_38057736_0 . . . 217
8.316 so_25902512_25902703_5 . . . 217
8.317 so_10618473_10618866_0 . . . 217
8.318 so_18543911_18544007_1 . . . 217
8.319 so_18543911_18544007_2 . . . 217
8.320 so_15959156_15959785_1 . . . 217
8.321 so_29172362_29172579_3 . . . 217
8.322 so_16505782_16509511_6 . . . 217
8.323 so_14390822_14390961_0 . . . 217
8.324 so_18551174_18552104_0 . . . 217
8.325 so_39980445_39980737_2 . . . 217
8.326 so_39980445_39980737_4 . . . 217
8.327 so_25882744_25882744_2 . . . 217
8.328 so_34972172_34973290_4 . . . 217
8.329 so_25926959_25926959_1 . . . 218
8.330 so_25926959_25926959_0 . . . 218
8.331 so_25926959_25927018_0 . . . 218
8.332 so_25926959_25927018_1 . . . 218
8.333 so_34972172_34973290_1 . . . 218
8.334 so_12879947_12880160_1 . . . 218
8.335 so_20129495_20129657_0 . . . 218
8.336 so_3607974_3608062_1 . . . 218
8.337 so_25987133_25987392_0 . . . 218
8.338 so_41273307_41273307_0 . . . 218
8.339 so_27470421_27471512_0 . . . 218
8.340 so_23872692_23872864_0 . . . 218
8.341 so_23872692_23872870_0 . . . 218
8.342 so_33197793_33197955_0 . . . 218
8.343 so_15850840_15850895_0 . . . 218
8.344 so_38128881_38128936_0 . . . 218
8.345 so_7197677_7197878_0 . . . 218
8.346 so_39860732_39861534_0 . . . 218
8.347 so_11555997_11556702_2 . . . 218
8.348 so_29454806_29454806_0 . . . 218
8.349 so_21127781_21127897_1 . . . 218
8.350 so_3920729_3925968_1 . . . 218
8.351 so_25956999_25957112_1 . . . 219
8.352 so_25956999_25957112_0 . . . 219
8.353 so_25956999_25956999_2 . . . 219
8.354 so_17995929_17996671_2 . . . 219
8.355 so_25416651_25416680_1 . . . 219
8.356 so_35837120_35837188_1 . . . 219
8.357 so_12879947_12880160_2 . . . 219
8.358 so_23772228_23786520_0 . . . 219
8.359 so_16739622_16740158_0 . . . 219
8.360 so_13756649_13756699_1 . . . 219
8.361 so_24259947_24263194_5 . . . 219
8.362 so_24259947_24263194_2 . . . 219
8.363 so_30737575_30737575_1 . . . 219
8.364 so_9649005_9649031_1 . . . 219

8.365 so_44247151_44288439_0 . . . 219
8.366 so_37493769_37514127_6 . . . 219
8.367 so_33005082_33005198_5 . . . 219
8.368 so_33005082_33005198_2 . . . 219
8.369 so_31387238_31387510_1 . . . 219
8.370 so_44143826_44143826_1 . . . 219
8.371 so_4937454_4938713_1 . . . 220
8.372 so_2322736_2322773_1 . . . 220
8.373 so_4706677_4707150_0 . . . 220
8.374 so_27594731_33502576_4 . . . 220
8.375 so_29896248_29901547_1 . . . 220
8.376 so_35708014_35708258_0 . . . 220
8.377 so_540748_540850_1 . . . 220
8.378 so_35920699_35922261_1 . . . 220
8.379 so_416345_36835586_2 . . . 220
8.380 so_32894131_32894445_0 . . . 220
8.381 so_44087145_44087209_1 . . . 220
8.382 so_44859450_44859450_0 . . . 220
8.383 so_44859450_44859591_7 . . . 220
8.384 so_44859450_44859795_2 . . . 220
8.385 so_37701321_37701760_0 . . . 220
8.386 so_51032_36835303_4 . . . 220
8.387 so_51032_36835303_0 . . . 220
8.388 so_7412274_7412299_1 . . . 220
8.389 so_35879510_35879765_0 . . . 220
8.390 so_5625600_5625782_1 . . . 220
8.391 so_18376783_18376899_5 . . . 220
8.392 so_18376783_18376783_0 . . . 220
8.393 so_15660279_15660477_0 . . . 220
8.394 so_2143906_2143995_1 . . . 220
8.395 so_9545208_9545279_1 . . . 220
8.396 so_9545208_9545279_0 . . . 220
8.397 so_13372173_13372928_0 . . . 220
8.398 so_40105746_40105806_0 . . . 220
8.399 so_39186871_39187930_0 . . . 220
8.400 so_33648117_33661933_1 . . . 220
8.401 so_27622520_27623121_0 . . . 220
8.402 so_13258665_13258818_0 . . . 220
8.403 so_10155415_10155608_0 . . . 220
8.404 so_26559758_26559758_0 . . . 220
8.405 so_40042078_40042198_0 . . . 220
8.406 so_35724423_35725486_2 . . . 220
8.407 so_23840421_23840460_1 . . . 220
8.408 so_25239203_25239492_0 . . . 220
8.409 so_36532687_36532978_1 . . . 220
8.410 so_15862549_15862607_0 . . . 220
8.411 so_2413097_2420230_3 . . . 220
8.412 so_4650974_4651412_0 . . . 220
8.413 so_13226942_13227752_0 . . . 220
8.414 so_13903524_13903587_0 . . . 220
8.415 so_35900465_35901059_0 . . . 220

8.416 so_43980505_43980657_1 . . . 220
8.417 so_43980505_43980711_0 . . . 220
8.418 so_43980505_43980668_0 . . . 220
8.419 so_39465725_39468475_0 . . . 220
8.420 so_24676824_24676976_0 . . . 220
8.421 so_11543625_11544332_2 . . . 220
8.422 so_11543625_11544332_3 . . . 220
8.423 so_27094506_27094672_1 . . . 220
8.424 so_34266308_34266765_0 . . . 220
8.425 so_28016179_28016189_1 . . . 220
8.426 so_8642741_8644519_0 . . . 220
8.427 so_30227277_30230036_2 . . . 220
8.428 so_6033970_6034240_0 . . . 220
8.429 so_43224984_43225060_0 . . . 220
8.430 so_34960981_34961245_0 . . . 220
8.431 so_22437809_22437836_0 . . . 220
8.432 so_29931718_29932110_0 . . . 220
8.433 so_29140223_29140340_1 . . . 220
8.434 so_5356686_5356754_0 . . . 220
8.435 so_12625679_12625732_0 . . . 220
8.436 so_47386833_47386921_0 . . . 220
8.437 so_41212904_41256180_1 . . . 220
8.438 so_28752247_28768636_2 . . . 220
8.439 so_14420908_14420908_8 . . . 220
8.440 so_15297644_15297695_0 . . . 220
8.441 so_1328223_1328246_0 . . . 220
8.442 so_4773334_4773450_0 . . . 220
8.443 so_4099239_4100093_0 . . . 220
8.444 so_27388480_27389156_0 . . . 220
8.445 so_27594273_27594665_0 . . . 220
8.446 so_27594273_27594665_4 . . . 220
8.447 so_27594273_27594665_3 . . . 220
8.448 so_28086502_28087648_3 . . . 220
8.449 so_28086502_28087648_4 . . . 220
8.450 so_15578298_15578298_0 . . . 220
8.451 so_15578298_15580072_2 . . . 220
8.452 so_14189585_14196449_1 . . . 220
8.453 so_15687905_15688216_0 . . . 220
8.454 so_13425618_13425708_0 . . . 220
8.455 so_38817156_38817548_0 . . . 220
8.456 so_12660845_12660897_2 . . . 220
8.457 so_7028174_7028681_0 . . . 220
8.458 so_39224689_39224809_1 . . . 220
8.459 so_47263793_47263794_1 . . . 220
8.460 so_9649005_9649031_2 . . . 220
8.461 so_27743745_27743745_5 . . . 220
8.462 so_27743745_27743745_3 . . . 220
8.463 so_27743745_27743745_0 . . . 220
8.464 so_28974316_28974609_3 . . . 220
8.465 so_36019049_36019088_1 . . . 220
8.466 so_26173403_26173403_0 . . . 220

8.467 so_17446220_17446559_3 . . . 220
8.468 so_1117873_1118283_0 . . . 220
8.469 so_1117873_1118283_1 . . . 220
8.470 so_38826640_38827256_0 . . . 220
8.471 so_18746237_18860724_0 . . . 220
8.472 so_46596259_46596449_0 . . . 220
8.473 so_25654186_25654229_3 . . . 220
8.474 so_22952480_23140119_1 . . . 220
8.475 so_22952480_23140119_0 . . . 220
8.476 so_33005082_33005198_9 . . . 220
8.477 so_38767993_38768590_0 . . . 220
8.478 so_29628571_29628826_0 . . . 221
8.479 so_17142130_17147477_4 . . . 221
8.480 so_5398930_5398968_0 . . . 221
8.481 so_25239203_25239492_1 . . . 221
8.482 so_30737575_30737832_1 . . . 221
8.483 so_14865174_14865174_0 . . . 221
8.484 so_4264304_4264495_0 . . . 221
8.485 so_29412146_29412417_0 . . . 221
8.486 so_16413651_16413734_1 . . . 221
8.487 so_16507334_16507538_1 . . . 221
8.488 so_31643245_31643380_0 . . . 221
8.489 so_18829274_18829776_0 . . . 221
8.490 so_8642741_8644519_1 . . . 221
8.491 so_24643825_24807560_0 . . . 221
8.492 so_24643825_24644237_0 . . . 221
8.493 so_5503473_5503643_0 . . . 221
8.494 so_26328077_26328406_0 . . . 221
8.495 so_22952480_23140119_3 . . . 221
8.496 so_22952480_23140119_2 . . . 221
8.497 so_24002299_24002611_1 . . . 221
8.498 so_28095723_28095884_0 . . . 221
8.499 so_27950769_27950769_0 . . . 221
8.500 so_34434675_34434814_0 . . . 221
8.501 so_24867294_24868095_1 . . . 221
8.502 so_24350354_24350505_0 . . . 221
8.503 so_31387238_31387510_0 . . . 221
8.504 so_16505782_16509511_0 . . . 221
8.505 so_16505782_16509511_2 . . . 221
8.506 so_17012648_17012659_0 . . . 221
8.507 so_7046647_7047049_1 . . . 221
8.508 so_2598178_2598300_0 . . . 221
8.509 so_29302272_29302329_1 . . . 221
8.510 so_29609081_29609337_0 . . . 221
8.511 so_9241255_9241363_2 . . . 221
8.512 so_28334419_28336771_0 . . . 221
8.513 so_19178920_19179047_0 . . . 221
8.514 so_42888031_42888065_0 . . . 221
8.515 so_20980192_20981414_0 . . . 221
8.516 so_8026579_8026593_0 . . . 221
8.517 so_15478101_15479358_0 . . . 221

8.518 so_17585191_17585259_0 . . . 221
8.519 so_16505782_16509511_1 . . . 221
8.520 so_7730874_7731106_1 . . . 221
8.521 so_7730874_9148408_0 . . . 221
8.522 so_41966731_41967148_2 . . . 221
8.523 so_17848983_17848983_0 . . . 221
8.524 so_13258665_13258818_1 . . . 221
8.525 so_32332666_32333629_0 . . . 221
8.526 so_36433402_36433544_0 . . . 221
8.527 so_8309698_8309763_0 . . . 221
8.528 so_30139580_30141622_1 . . . 221
8.529 so_17446220_17446559_1 . . . 221
8.530 so_47473986_47478004_1 . . . 221
8.531 so_45074835_45076265_0 . . . 221
8.532 so_20840822_20840915_0 . . . 221
8.533 so_16411599_16411743_0 . . . 221
8.534 so_14675908_14675993_1 . . . 222
8.535 so_8776810_8776886_1 . . . 222
8.536 so_39224689_39224809_0 . . . 222
8.537 so_32809322_32809553_0 . . . 222
8.538 so_25730544_25730902_0 . . . 222
8.539 so_4417219_4417294_0 . . . 222
8.540 so_47230544_47230715_1 . . . 222
8.541 so_32194151_32194324_2 . . . 222
8.542 so_19269438_19269611_0 . . . 222
8.543 so_4523617_4574567_0 . . . 222
8.544 so_28714018_28714908_2 . . . 222
8.545 so_13766559_13767066_0 . . . 222
8.546 so_16698813_16698959_1 . . . 223
8.547 so_14675908_14675993_0 . . . 223
8.548 so_18740917_18741089_0 . . . 223
8.549 so_13425618_13425747_0 . . . 223
8.550 so_13425618_13425708_1 . . . 223
8.551 so_38131945_38131963_0 . . . 223
8.552 so_29665532_29665567_0 . . . 223
8.553 so_37846565_37846719_1 . . . 223
8.554 so_26422189_26422189_0 . . . 223
8.555 so_16698813_16698959_4 . . . 223
8.556 so_16698813_16698959_5 . . . 223
8.557 so_14675908_14675952_1 . . . 223
8.558 so_37846565_37846719_3 . . . 223
8.559 so_37846565_37846719_4 . . . 223
8.560 so_27909109_28764887_12 . . . 223
8.561 so_27909109_28764887_13 . . . 223
8.562 so_44143826_44143826_0 . . . 223
8.563 so_44143826_44143991_0 . . . 223
8.564 so_27686606_27686690_0 . . . 223
8.565 so_29098835_29098959_0 . . . 223
8.566 so_18313509_18313750_1 . . . 223
8.567 so_13713916_13713944_0 . . . 223
8.568 so_32105975_32105997_0 . . . 223

8.569 so_32359338_32359338_0 . . . 223
8.570 so_37428228_37428357_0 . . . 223
8.571 so_6805507_6805528_0 . . . 223
8.572 so_26721092_26721182_0 . . . 223
8.573 so_14189585_14196449_2 . . . 223
8.574 so_38125820_38126010_0 . . . 223
8.575 so_35980469_35983656_1 . . . 223
8.576 so_19483804_19483845_1 . . . 223
8.577 so_47330640_47330728_0 . . . 223
8.578 so_27813482_27813711_1 . . . 223
8.579 so_27813482_27813711_0 . . . 223
8.580 so_28663851_28664441_0 . . . 223
8.581 so_16256899_16256919_0 . . . 223
8.582 so_7518744_7518785_0 . . . 223
8.583 so_20550459_20550523_0 . . . 223
8.584 so_23333949_23336823_0 . . . 223
8.585 so_31678245_31695350_1 . . . 223
8.586 so_26352319_26352410_0 . . . 223
8.587 so_26352319_26352410_1 . . . 223
8.588 so_6708726_6708956_0 . . . 223
8.589 so_5637679_5637862_0 . . . 223
8.590 so_29098835_29098959_1 . . . 223
8.591 so_31528710_31528794_0 . . . 223
8.592 so_29871138_29957648_4 . . . 223
8.593 so_44818513_44818513_0 . . . 223
8.594 so_27594710_27594839_0 . . . 223
8.595 so_3858344_3858365_0 . . . 223
8.596 so_24017466_24026094_3 . . . 223
8.597 so_35640067_35640129_0 . . . 223
8.598 so_4120242_4120429_0 . . . 224
8.599 so_25097032_25097443_0 . . . 224
8.600 so_38131945_38131963_1 . . . 224
8.601 so_851584_851760_0 . . . 224
8.602 so_13484308_13484670_0 . . . 224
8.603 so_26422189_26422189_1 . . . 224
8.604 so_45890725_45890840_0 . . . 224
8.605 so_45890725_45890840_1 . . . 224
8.606 so_45890725_45890841_0 . . . 224
8.607 so_32105975_32105997_1 . . . 224
8.608 so_6210450_6210467_0 . . . 224
8.609 so_24471511_24471650_0 . . . 224
8.610 so_43393787_43394295_0 . . . 224
8.611 so_26687852_26687972_0 . . . 224
8.612 so_2479846_2479899_0 . . . 224
8.613 so_31678245_31695350_2 . . . 224
8.614 so_5637679_5638060_0 . . . 224
8.615 so_25647199_25647443_0 . . . 224
8.616 so_23725995_23726055_0 . . . 225
8.617 so_13678627_13678730_0 . . . 225
8.618 so_12903724_12903763_0 . . . 225
8.619 so_12903724_12903763_2 . . . 225

8.620 so_26999739_26999944_0 . . . 225
8.621 so_27807215_27807289_0 . . . 225
8.622 so_27807215_27807281_0 . . . 225
8.623 so_36053305_36053478_1 . . . 225
8.624 so_9116036_9116036_0 . . . 225
8.625 so_9116036_9116162_1 . . . 225
8.626 so_38471737_38471836_1 . . . 225
8.627 so_38471737_38471802_1 . . . 225
8.628 so_23725995_23726055_1 . . . 225
8.629 so_13899440_13899445_0 . . . 225
8.630 so_30539849_30539954_0 . . . 225
8.631 so_13425618_13425747_1 . . . 225
8.632 so_35724423_35725013_0 . . . 225
8.633 so_13678627_13678730_1 . . . 225
8.634 so_13678627_13678627_0 . . . 225
8.635 so_27018251_27018334_0 . . . 225
8.636 so_27018251_27018334_2 . . . 225
8.637 so_27018251_27018334_1 . . . 225
8.638 so_29924617_29924809_0 . . . 225
8.639 so_37144183_37144183_1 . . . 225
8.640 so_4539406_6363678_0 . . . 225
8.641 so_4539406_4539444_0 . . . 225
8.642 so_11060968_11061103_1 . . . 225
8.643 so_11167062_11167081_0 . . . 225
8.644 so_5273144_5273165_1 . . . 225
8.645 so_1880866_1880877_0 . . . 225
8.646 so_33086820_33087331_0 . . . 225
8.647 so_41908781_41909021_0 . . . 225
8.648 so_26999739_26999944_1 . . . 225
8.649 so_37335383_37335422_0 . . . 225
8.650 so_27807215_27807289_2 . . . 225
8.651 so_27807215_27807289_3 . . . 225
8.652 so_36053305_36053478_0 . . . 225
8.653 so_39313362_39313376_0 . . . 225
8.654 so_8851383_8851493_0 . . . 225
8.655 so_5625677_5625778_1 . . . 225
8.656 so_5625677_5625677_0 . . . 225
8.657 so_5625677_5625778_0 . . . 225
8.658 so_25725936_25725936_0 . . . 225
8.659 so_39397028_39397256_0 . . . 225
8.660 so_15440833_15440852_1 . . . 225
8.661 so_2576411_2576428_0 . . . 225
8.662 so_27866797_27866956_0 . . . 225
8.663 so_21133717_21134247_0 . . . 225
8.664 so_35819580_35819610_0 . . . 225
8.665 so_9146732_9146789_0 . . . 225
8.666 so_12847456_12847480_0 . . . 225
8.667 so_21559238_21559430_0 . . . 225
8.668 so_24393489_24393520_0 . . . 225
8.669 so_27947352_27947714_0 . . . 225
8.670 so_42363040_42363605_0 . . . 225

8.671 so_26422189_26422189_2 . . . 225
8.672 so_1625105_17672456_0 . . . 225
8.673 so_37919291_37919534_0 . . . 225
8.674 so_37919291_37919414_0 . . . 225
8.675 so_20227620_20227970_0 . . . 225
8.676 so_40513156_40513319_2 . . . 225
8.677 so_40513156_40513319_4 . . . 225
8.678 so_40513156_40513319_5 . . . 225
8.679 so_37689461_37689461_0 . . . 225
8.680 so_26658776_26658974_0 . . . 225
8.681 so_9227333_9227388_1 . . . 225
8.682 so_2934904_2934917_0 . . . 225
8.683 so_2934904_47605559_0 . . . 225
8.684 so_7112282_7112409_1 . . . 225
8.685 so_7112282_7112409_0 . . . 225
8.686 so_27974653_27974956_1 . . . 226
8.687 so_12775791_12775791_0 . . . 226
8.688 so_12775791_12775833_0 . . . 226
8.689 so_5273144_5273165_0 . . . 226
8.690 so_44122476_44122589_0 . . . 226
8.691 so_44122476_44122589_1 . . . 226
8.692 so_44122476_44122589_2 . . . 226
8.693 so_44122476_44122476_0 . . . 226
8.694 so_12245019_12245026_0 . . . 226
8.695 so_12245019_12245105_0 . . . 226
8.696 so_12245019_12245090_0 . . . 226
8.697 so_51032_51044_0 . . . 226
8.698 so_22895156_22895275_0 . . . 226
8.699 so_39283164_39283179_1 . . . 226
8.700 so_29293180_29293313_0 . . . 226
8.701 so_23727952_23727984_0 . . . 226
8.702 so_37532254_37532323_0 . . . 226
8.703 so_25994594_25994811_0 . . . 226
8.704 so_6464404_6464440_0 . . . 226
8.705 so_3533589_3533692_0 . . . 226
8.706 Function definitions . . . 226
8.707 so_18861184_18861284_0 . . . 226
8.708 so_36295738_37138998_9 . . . 226
8.709 so_36295738_37138998_13 . . . 226
8.710 so_22895078_22896479_0 . . . 226
8.711 so_22895078_22896479_1 . . . 226
8.712 so_32866810_32867695_0 . . . 226
8.713 so_13993798_13995360_2 . . . 227
8.714 so_14015142_14015173_0 . . . 227
8.715 so_21055039_21055215_0 . . . 227
8.716 so_27408928_27409772_4 . . . 227
8.717 so_26976567_26976651_0 . . . 227
8.718 so_46873142_46873181_1 . . . 227
8.719 so_16598320_16598329_2 . . . 227
8.720 so_18746237_18860724_1 . . . 227
8.721 so_2319663_31159628_0 . . . 227

8.722 so_47230544_47230715_0 . . . 227
8.723 so_14123347_14123347_0 . . . 227
8.724 so_14164161_14164161_2 . . . 227
8.725 so_23298586_23298844_1 . . . 227
8.726 so_23298586_23298844_0 . . . 227
8.727 so_7649944_7650065_0 . . . 227
8.728 so_6151558_6151580_0 . . . 227
8.729 so_6151558_6151580_2 . . . 227
8.730 so_27295881_27296006_0 . . . 227
8.731 so_13655198_13655252_0 . . . 228
8.732 so_3105798_26985436_3 . . . 228
8.733 so_41551545_41551678_2 . . . 228
8.734 so_36223198_36223967_4 . . . 228
8.735 so_27233769_27233769_2 . . . 228
8.736 so_27233769_27233769_4 . . . 228
8.737 so_27233769_27233848_0 . . . 228
8.738 so_35642814_35643113_0 . . . 228
8.739 so_18290523_18290839_2 . . . 228
8.740 so_18290523_18290839_4 . . . 228
8.741 so_10370713_10376279_0 . . . 228
8.742 so_12793412_12793431_2 . . . 228
8.743 so_46559313_46559427_2 . . . 228
8.744 so_37191293_37191397_4 . . . 228
8.745 so_37191293_37191397_2 . . . 228
8.746 so_20610658_20616637_1 . . . 228
8.747 so_10731131_10731415_0 . . . 228
8.748 so_36161188_36161478_0 . . . 228
8.749 so_47018465_47018614_0 . . . 228
8.750 so_20810378_20810439_2 . . . 228
8.751 so_28660945_28661719_0 . . . 228
8.752 so_33180114_33180114_0 . . . 228
8.753 so_25465379_25465850_1 . . . 228
8.754 so_29850923_29854565_0 . . . 228
8.755 so_29483120_29483269_1 . . . 228
8.756 so_29483120_29483269_0 . . . 228
8.757 so_36161188_36161478_2 . . . 228
8.758 so_36161188_36161478_1 . . . 228
8.759 so_12754886_12755043_1 . . . 228
8.760 so_12073556_12074507_0 . . . 228
8.761 so_27050187_27050336_2 . . . 228
8.762 so_26954437_26954437_0 . . . 228
8.763 so_30096911_30097049_5 . . . 228
8.764 so_16076468_16076878_1 . . . 228
8.765 so_24943665_24943751_1 . . . 228
8.766 so_21900237_21900737_0 . . . 228
8.767 so_22711901_22711901_0 . . . 228
8.768 so_26699720_26700151_9 . . . 228
8.769 so_26699720_26699720_10 . . . 228
8.770 so_17221668_17222286_4 . . . 228
8.771 so_17221668_17222148_0 . . . 228
8.772 so_37618213_37618407_1 . . . 228

8.773 so_37618213_37618395_0 . . . 228
8.774 so_28555280_28555380_0 . . . 228
8.775 so_23882409_23883036_1 . . . 228
8.776 so_23882409_23883035_18 . . . 228
8.777 so_22357887_22360283_1 . . . 228
8.778 so_40195444_40195598_0 . . . 228
8.779 so_29759441_29759556_5 . . . 228
8.780 so_16897845_16897845_0 . . . 228
8.781 so_28963577_28965673_0 . . . 228
8.782 so_18216313_18216727_1 . . . 228
8.783 so_7169658_7169675_0 . . . 228
8.784 so_46484307_46484307_4 . . . 228
8.785 so_44886942_45944822_0 . . . 228
8.786 so_36357172_36357419_0 . . . 228
8.787 so_36357172_36357419_3 . . . 228
8.788 so_36357172_36357434_0 . . . 228
8.789 so_20936574_20936695_4 . . . 228
8.790 so_38929950_38931484_0 . . . 228
8.791 so_33146096_33146987_0 . . . 228
8.792 so_23007358_23007559_0 . . . 228
8.793 so_31866437_31866878_4 . . . 228
8.794 so_27233769_27233769_8 . . . 229
8.795 so_19265920_19266242_2 . . . 229
8.796 so_37276413_37277533_5 . . . 229
8.797 so_26994755_26994929_1 . . . 229
8.798 so_20816117_20816214_0 . . . 229
8.799 so_36603628_36603734_1 . . . 229
8.800 so_17786033_17786531_0 . . . 229
8.801 so_40608180_40608703_0 . . . 229
8.802 so_16076468_16076878_6 . . . 229
8.803 so_16703798_16703973_1 . . . 229
8.804 so_34186324_34211359_0 . . . 229
8.805 so_26569239_26569369_0 . . . 229
8.806 so_26569239_26569369_1 . . . 229
8.807 so_19265920_19266242_4 . . . 229
8.808 so_32716589_32716722_0 . . . 229
8.809 so_29879834_29880264_0 . . . 229
8.810 so_37276413_37277533_6 . . . 229
8.811 so_46314709_46314790_0 . . . 229
8.812 so_7935639_7935954_1 . . . 229
8.813 so_14085553_14085599_1 . . . 229
8.814 so_14085553_14085774_0 . . . 229
8.815 so_14085620_14086455_2 . . . 229
8.816 so_14085620_14086455_1 . . . 229
8.817 so_14085620_14086455_0 . . . 229
8.818 so_32299171_32299536_7 . . . 230
8.819 so_19265920_19266242_3 . . . 230
8.820 so_14085553_14085599_0 . . . 230
8.821 Initializers . . . 230
8.822 so_30518780_30518890_0 . . . 230
8.823 so_10614728_10615016_0 . . . 230

8.824 so_19092035_19092298_5 . . . 230
8.825 so_26182734_26183427_1 . . . 230
8.826 so_46070793_46071137_0 . . . 230
8.827 so_12306885_12306885_1 . . . 230
8.828 so_26323471_26323471_0 . . . 230
8.829 so_23720258_23720289_1 . . . 230
8.830 so_24435954_24436482_1 . . . 230
8.831 so_25864206_25864379_0 . . . 230
8.832 so_6337426_6337447_1 . . . 230
8.833 so_24771192_24771913_1 . . . 231
8.834 so_24814912_24814938_0 . . . 231
8.835 so_33126333_33126532_2 . . . 231
8.836 so_33126333_33126532_3 . . . 231
8.837 so_33126333_33126532_1 . . . 231
8.838 so_42074424_42074466_1 . . . 231
8.839 so_27642882_27643146_4 . . . 231
8.840 so_27642882_27643146_5 . . . 231
8.841 so_27642882_27643146_8 . . . 231
8.842 so_27642882_27643207_2 . . . 231
8.843 so_27642882_27643207_3 . . . 231
8.844 so_27642882_27643146_9 . . . 231
8.845 so_27642882_27643146_1 . . . 231
8.846 so_27642882_27643146_6 . . . 231
8.847 so_27642882_27643146_0 . . . 231
8.848 so_27642882_27643146_3 . . . 231
8.849 so_27642882_27643146_7 . . . 231
8.850 so_27642882_27643146_2 . . . 231
8.851 so_11812687_11812718_4 . . . 231
8.852 so_28408911_28409778_5 . . . 231
8.853 so_19575202_19575287_15 . . . 231
8.854 so_19575202_19575287_14 . . . 231
8.855 so_19575202_19575287_12 . . . 231
8.856 so_19575202_19575287_13 . . . 231
8.857 so_19575202_19575287_11 . . . 231
8.858 so_19575202_19575287_9 . . . 231
8.859 so_25903968_25904184_11 . . . 231
8.860 so_25903968_25904184_10 . . . 231
8.861 so_26648817_26649961_4 . . . 231
8.862 so_11164394_11164416_2 . . . 231
8.863 so_18308941_18308941_3 . . . 231
8.864 so_453432_453462_0 . . . 231
8.865 so_39026871_39027160_0 . . . 231
8.866 so_42361656_42367505_2 . . . 231
8.867 so_7084831_7085249_4 . . . 231
8.868 so_7084831_7085249_2 . . . 231
8.869 so_7084831_7085249_1 . . . 231
8.870 so_7084831_7085249_5 . . . 231
8.871 so_7084831_7085249_3 . . . 231
8.872 so_7084831_7085249_0 . . . 231
8.873 so_20788960_20789054_2 . . . 231
8.874 so_20788960_20789054_1 . . . 231

8.875 so_31510673_31510673_0 . . . 231
8.876 so_8943261_8943287_11 . . . 231
8.877 so_8943261_8943287_8 . . . 231
8.878 so_6284507_6284695_2 . . . 231
8.879 so_38131945_38131963_6 . . . 231
8.880 so_27935026_30153809_0 . . . 231
8.881 so_27382036_27388378_1 . . . 231
8.882 so_27382036_27388378_2 . . . 231
8.883 so_5560363_5560415_1 . . . 231
8.884 so_27486581_27486581_10 . . . 231
8.885 so_29765961_29767272_2 . . . 231
8.886 so_11058422_11058463_1 . . . 231
8.887 so_11058422_11058463_2 . . . 231
8.888 so_33913572_33913851_3 . . . 231
8.889 so_5602030_5602057_2 . . . 231
8.890 so_29109654_29109746_1 . . . 231
8.891 so_32377780_32377854_7 . . . 231
8.892 so_32377780_32377854_6 . . . 231
8.893 so_26793250_26793467_2 . . . 231
8.894 so_25043717_25043996_1 . . . 231
8.895 so_14259602_14259765_3 . . . 231
8.896 so_27484483_27484598_6 . . . 231
8.897 so_34856965_34857406_3 . . . 231
8.898 so_34856965_34857406_6 . . . 231
8.899 so_34856965_34857406_1 . . . 231
8.900 so_34856965_34857406_4 . . . 231
8.901 so_47437349_47440920_7 . . . 231
8.902 so_47437349_47440920_5 . . . 231
8.903 so_47437349_47440920_6 . . . 231
8.904 so_47437349_47440920_0 . . . 231
8.905 so_47437349_47440920_3 . . . 231
8.906 so_47437349_47440920_4 . . . 231
8.907 so_47437349_47440920_1 . . . 231
8.908 so_32648805_32652158_4 . . . 231
8.909 so_21964120_21964142_3 . . . 231
8.910 so_21964120_21964142_5 . . . 231
8.911 so_26699720_26700151_11 . . . 231
8.912 so_26699720_26699720_7 . . . 231
8.913 so_26699720_26699720_6 . . . 231
8.914 so_26699720_26700151_10 . . . 231
8.915 so_26699720_26699720_8 . . . 231
8.916 so_5610352_5610375_3 . . . 231
8.917 so_19658609_19658743_2 . . . 231
8.918 so_19658609_19658743_1 . . . 231
8.919 so_25528245_25528459_1 . . . 231
8.920 so_25528245_25528459_2 . . . 231
8.921 so_40943080_40943130_3 . . . 231
8.922 so_40943080_40943130_4 . . . 231
8.923 so_29291371_29292092_4 . . . 231
8.924 so_21263381_21263381_0 . . . 231
8.925 so_3580789_3580876_11 . . . 231

8.926 so_3580789_3580876_8 . . . 231
8.927 so_3580789_3580876_10 . . . 231
8.928 so_3580789_3580876_6 . . . 231
8.929 so_3580789_3580876_9 . . . 231
8.930 so_1628311_1628396_2 . . . 231
8.931 so_17257660_17257681_4 . . . 231
8.932 so_17257660_17257681_2 . . . 231
8.933 so_17257660_17257681_3 . . . 231
8.934 so_17257660_17257681_5 . . . 231
8.935 so_17257660_17257681_0 . . . 231
8.936 so_42534992_42536286_1 . . . 231
8.937 so_42534992_42535090_2 . . . 231
8.938 so_6511971_6512058_2 . . . 231
8.939 so_9342758_9342993_3 . . . 231
8.940 so_9342758_9342993_4 . . . 231
8.941 so_9342758_9342993_2 . . . 231
8.942 so_9342758_9342993_1 . . . 231
8.943 so_9342758_9342993_0 . . . 231
8.944 so_9342758_9342993_5 . . . 231
8.945 so_40098080_40098140_5 . . . 231
8.946 so_1613341_1613383_0 . . . 231
8.947 so_14829158_14829622_12 . . . 231
8.948 so_14829158_14829622_15 . . . 231
8.949 so_14829158_14829622_9 . . . 231
8.950 so_14829158_14829622_14 . . . 231
8.951 so_14829158_14829622_13 . . . 231
8.952 so_14829158_14829622_11 . . . 231
8.953 so_15548813_15548855_2 . . . 231
8.954 so_15548813_15548978_7 . . . 231
8.955 so_15548813_15548978_4 . . . 231
8.956 so_15548813_15548873_7 . . . 231
8.957 so_47096920_47097250_2 . . . 231
8.958 so_19121890_19122919_3 . . . 231
8.959 so_3553559_3553682_3 . . . 231
8.960 so_3553559_3553682_1 . . . 231
8.961 so_3553559_3553682_2 . . . 231
8.962 so_3553559_3553682_4 . . . 231
8.963 so_3553559_3553682_0 . . . 231
8.964 so_17257828_17257838_15 . . . 231
8.965 so_17257828_17257838_13 . . . 231
8.966 so_17257828_17257838_14 . . . 231
8.967 so_17257828_17257838_16 . . . 231
8.968 so_17257828_17257838_11 . . . 231
8.969 so_22849688_22849688_6 . . . 231
8.970 so_22849688_22849688_3 . . . 231
8.971 so_22849688_22849688_1 . . . 231
8.972 so_22849688_22849688_5 . . . 231
8.973 so_22849688_22849688_4 . . . 231
8.974 so_9025792_9025981_3 . . . 231
8.975 so_15080742_15080926_14 . . . 231
8.976 so_15080742_15080926_15 . . . 231

8.977 so_15080742_15080926_16 . . . 231
8.978 so_15080742_15080926_17 . . . 231
8.979 so_15080742_15080926_18 . . . 231
8.980 so_15080742_15080926_12 . . . 231
8.981 so_16339811_16340318_2 . . . 231
8.982 so_18370170_18370170_0 . . . 231
8.983 so_5950144_5950176_2 . . . 231
8.984 so_4523959_4524064_1 . . . 231
8.985 so_9173322_9173805_1 . . . 231
8.986 so_1126107_1126995_2 . . . 231
8.987 so_25438023_25438102_8 . . . 231
8.988 so_25438023_25438102_5 . . . 231
8.989 so_25438023_25438102_6 . . . 231
8.990 so_25438023_25438102_4 . . . 231
8.991 so_16780786_16781321_0 . . . 231
8.992 so_15149363_15149482_2 . . . 231
8.993 so_15149363_15149482_1 . . . 231
8.994 so_11812687_11812718_0 . . . 231
8.995 so_11812687_11812718_1 . . . 231
8.996 so_20816117_20816117_0 . . . 231
8.997 so_20816117_20816117_1 . . . 231
8.998 so_20638968_20639122_2 . . . 231
8.999 so_8202571_8202615_0 . . . 231
8.1000 so_20576162_20576364_3 . . . 231
8.1001 so_33079486_33079574_4 . . . 231
8.1002 so_19575202_19575287_4 . . . 231
8.1003 so_17106389_17106428_7 . . . 231
8.1004 so_17106389_17106428_5 . . . 231
8.1005 so_17106389_17106428_3 . . . 231
8.1006 so_17106389_17106428_4 . . . 231
8.1007 so_20909693_20909693_0 . . . 231
8.1008 so_17276416_17276482_5 . . . 231
8.1009 so_17276416_17276482_7 . . . 231
8.1010 so_17276416_17276482_4 . . . 231
8.1011 so_17276416_17276482_3 . . . 231
8.1012 so_18308941_18308941_4 . . . 231
8.1013 so_453432_453462_1 . . . 231
8.1014 so_24122006_24122006_1 . . . 231
8.1015 so_37655925_37655925_3 . . . 231
8.1016 so_13593165_13593507_2 . . . 231
8.1017 so_13593165_13593507_3 . . . 231
8.1018 so_13593165_13593507_1 . . . 231
8.1019 so_33813929_33813930_4 . . . 231
8.1020 so_7084831_7085249_8 . . . 231
8.1021 so_7084831_7085249_7 . . . 231
8.1022 so_7084831_7085249_11 . . . 231
8.1023 so_7084831_7085249_6 . . . 231
8.1024 so_25501583_25501734_3 . . . 231
8.1025 so_30603512_30603673_5 . . . 231
8.1026 so_30603512_30603673_4 . . . 231
8.1027 so_30603512_30603673_3 . . . 231

8.1028 so_8697105_8697182_1 . . . 231
8.1029 so_26077807_26077807_0 . . . 231
8.1030 so_26077807_26077807_3 . . . 231
8.1031 so_26077807_26077807_1 . . . 231
8.1032 so_33809441_33809698_1 . . . 231
8.1033 so_38131945_38131963_5 . . . 231
8.1034 so_27258364_27329244_3 . . . 231
8.1035 so_23987515_23987926_4 . . . 231
8.1036 so_24943665_24943751_0 . . . 231
8.1037 so_24943665_24946007_0 . . . 231
8.1038 so_44853774_44853774_1 . . . 231
8.1039 so_14782176_14782518_3 . . . 231
8.1040 so_30847558_30847679_0 . . . 231
8.1041 so_27117169_27117169_6 . . . 231
8.1042 so_7760291_7760316_1 . . . 231
8.1043 so_7760291_7760316_0 . . . 231
8.1044 so_7760291_7760316_2 . . . 231
8.1045 so_15030379_15030445_1 . . . 231
8.1046 so_15030379_15030445_3 . . . 231
8.1047 so_15030379_15030445_2 . . . 231
8.1048 so_29765961_29767272_3 . . . 231
8.1049 so_29765961_29766171_1 . . . 231
8.1050 so_14895428_14895493_3 . . . 231
8.1051 so_12164061_12164116_3 . . . 231
8.1052 so_19449330_19450808_1 . . . 231
8.1053 so_29109654_29109746_3 . . . 231
8.1054 so_32377780_32377854_1 . . . 231
8.1055 so_27200284_27200678_2 . . . 231
8.1056 so_11729603_11729729_1 . . . 231
8.1057 so_21900237_21900356_0 . . . 231
8.1058 so_21900237_21900737_1 . . . 231
8.1059 so_21900237_21900237_0 . . . 231
8.1060 so_21900237_21901170_0 . . . 231
8.1061 so_31389454_31389824_0 . . . 231
8.1062 so_27484483_27484598_1 . . . 231
8.1063 so_30180417_30180440_0 . . . 231
8.1064 so_23334719_23335067_11 . . . 231
8.1065 so_23334719_23335067_5 . . . 231
8.1066 so_23334719_23335067_4 . . . 231
8.1067 so_23334719_23335067_3 . . . 231
8.1068 so_34856965_34857406_9 . . . 231
8.1069 so_34856965_34857406_7 . . . 231
8.1070 so_34856965_34857406_8 . . . 231
8.1071 so_31165342_31165342_0 . . . 231
8.1072 so_31165342_31165342_2 . . . 231
8.1073 so_7411515_7411708_1 . . . 231
8.1074 so_7411515_7411515_0 . . . 231
8.1075 so_46909105_46909166_0 . . . 231
8.1076 so_46909105_46909166_2 . . . 231
8.1077 so_47437349_47440920_8 . . . 231
8.1078 so_47437349_47440920_10 . . . 231

8.1079 so_47437349_47440920_9 . . . 231
8.1080 so_42753585_42754529_0 . . . 231
8.1081 so_24333458_24333458_0 . . . 231
8.1082 so_24333458_24333884_4 . . . 231
8.1083 so_27668269_27668438_6 . . . 231
8.1084 so_25999309_25999448_3 . . . 231
8.1085 so_25999309_25999448_2 . . . 231
8.1086 so_25999309_25999448_1 . . . 231
8.1087 so_25999309_25999455_0 . . . 231
8.1088 so_42251142_42251142_1 . . . 231
8.1089 so_42251142_42252328_3 . . . 231
8.1090 so_8952284_8952318_0 . . . 231
8.1091 so_31766613_31766689_1 . . . 231
8.1092 so_25278066_25278125_0 . . . 231
8.1093 so_20849454_20849500_3 . . . 231
8.1094 so_13576055_23698999_1 . . . 231
8.1095 so_27699118_27699260_13 . . . 231
8.1096 so_27699118_27699260_10 . . . 231
8.1097 so_27699118_27699260_8 . . . 231
8.1098 so_22795037_22795079_1 . . . 231
8.1099 so_3580789_3580876_2 . . . 231
8.1100 so_1628311_1628396_3 . . . 231
8.1101 so_25395297_25395408_1 . . . 231
8.1102 so_25395297_25395408_2 . . . 231
8.1103 so_25395297_25395408_3 . . . 231
8.1104 so_17257660_17257681_7 . . . 231
8.1105 so_46631675_46631769_3 . . . 231
8.1106 so_46631675_46631769_2 . . . 231
8.1107 so_46631675_46631769_1 . . . 231
8.1108 so_33239935_33240173_0 . . . 231
8.1109 so_8256293_8256293_1 . . . 231
8.1110 so_8256293_8256369_1 . . . 231
8.1111 so_8256293_8256293_0 . . . 231
8.1112 so_8256293_8256369_3 . . . 231
8.1113 so_8256293_8256369_2 . . . 231
8.1114 so_8256293_8256293_2 . . . 231
8.1115 so_6511971_6512058_3 . . . 231
8.1116 so_23332744_23333184_0 . . . 231
8.1117 so_9342758_9342993_7 . . . 231
8.1118 so_9342758_9342993_8 . . . 231
8.1119 so_9342758_9342993_6 . . . 231
8.1120 so_9342758_9342993_10 . . . 231
8.1121 so_9342758_9342993_12 . . . 231
8.1122 so_28337951_28337951_0 . . . 231
8.1123 so_28337951_28338123_0 . . . 231
8.1124 so_30172483_30172740_2 . . . 231
8.1125 so_30172483_30172483_1 . . . 231
8.1126 so_30172483_30172733_0 . . . 231
8.1127 so_22265255_22265499_2 . . . 231
8.1128 so_22265255_22265499_4 . . . 231
8.1129 so_22265255_22265499_3 . . . 231

8.1130 so_27584999_27585055_0 . . . 231
8.1131 so_27584999_27585071_0 . . . 231
8.1132 so_1613341_1613383_1 . . . 231
8.1133 so_8092670_8092791_2 . . . 231
8.1134 so_8092670_8092791_3 . . . 231
8.1135 so_14829158_14829622_4 . . . 231
8.1136 so_14829158_14829622_19 . . . 231
8.1137 so_14829158_14829622_21 . . . 231
8.1138 so_14829158_14829622_20 . . . 231
8.1139 so_15548813_15548978_3 . . . 231
8.1140 so_29683381_29683381_0 . . . 231
8.1141 so_4950125_4950186_2 . . . 231
8.1142 so_4950125_4950186_1 . . . 231
8.1143 so_4950125_4950186_0 . . . 231
8.1144 so_2836939_2836939_7 . . . 231
8.1145 so_2836939_2836939_3 . . . 231
8.1146 so_2836939_2836939_4 . . . 231
8.1147 so_19121890_19122919_8 . . . 231
8.1148 so_17257828_17257838_7 . . . 231
8.1149 so_27505539_27506091_1 . . . 231
8.1150 so_27505539_27506091_0 . . . 231
8.1151 so_46427709_46430666_2 . . . 231
8.1152 so_46427709_46430666_0 . . . 231
8.1153 so_22849688_22849688_7 . . . 231
8.1154 so_22849688_22849688_12 . . . 231
8.1155 so_22849688_22849688_9 . . . 231
8.1156 so_22849688_22849688_10 . . . 231
8.1157 so_22849688_22849688_8 . . . 231
8.1158 so_46287633_46288718_0 . . . 231
8.1159 so_18166885_18166885_0 . . . 231
8.1160 so_18166885_18166923_0 . . . 231
8.1161 so_18166885_18166896_0 . . . 231
8.1162 so_9502810_9502847_1 . . . 231
8.1163 so_22233148_22233148_2 . . . 231
8.1164 so_22233148_22233148_1 . . . 231
8.1165 so_22233148_22233148_0 . . . 231
8.1166 so_27349679_27349679_2 . . . 231
8.1167 so_27349679_27349679_0 . . . 231
8.1168 so_27349679_27349679_5 . . . 231
8.1169 so_27349679_27349679_1 . . . 231
8.1170 so_27349679_27349771_0 . . . 231
8.1171 so_15080742_15080926_7 . . . 231
8.1172 so_16339811_16340318_3 . . . 231
8.1173 so_18901315_18902462_6 . . . 231
8.1174 so_35383453_35383576_15 . . . 231
8.1175 so_35383453_35383576_14 . . . 231
8.1176 so_11993134_11993284_1 . . . 231
8.1177 so_14659752_14659931_3 . . . 231
8.1178 so_14659752_14659931_6 . . . 231
8.1179 so_14659752_14659881_1 . . . 231
8.1180 so_3895647_3895647_1 . . . 231

8.1181 so_1751378_1751546_2 . . . 231
8.1182 so_4523959_4524064_0 . . . 231
8.1183 so_37909549_37909549_0 . . . 231
8.1184 so_37909549_37909950_0 . . . 231
8.1185 so_17477573_17477638_0 . . . 231
8.1186 so_9173322_9173805_2 . . . 231
8.1187 so_9173322_9173805_0 . . . 231
8.1188 so_9173322_9173805_3 . . . 231
8.1189 so_1126107_1126995_1 . . . 231
8.1190 so_1126107_1126995_4 . . . 231
8.1191 so_25299995_25300292_5 . . . 231
8.1192 so_15149363_15149482_4 . . . 231
8.1193 so_42074424_42074466_0 . . . 231
8.1194 so_20255173_20255173_0 . . . 231
8.1195 so_27642882_27643207_1 . . . 231
8.1196 so_11812687_11812718_2 . . . 231
8.1197 so_11812687_11812718_3 . . . 231
8.1198 so_20638968_20639122_1 . . . 231
8.1199 so_28408911_28409778_9 . . . 231
8.1200 so_28408911_28409778_4 . . . 231
8.1201 so_20576162_20576364_2 . . . 231
8.1202 so_33079486_33079574_3 . . . 231
8.1203 so_19575202_19575287_6 . . . 231
8.1204 so_19575202_19575287_3 . . . 231
8.1205 so_19575202_19575287_5 . . . 231
8.1206 so_37516576_37516764_1 . . . 231
8.1207 so_37516576_37516764_0 . . . 231
8.1208 so_25903968_25904184_9 . . . 231
8.1209 so_21940769_21940769_0 . . . 231
8.1210 so_11164394_11164416_1 . . . 231
8.1211 so_7084831_7085249_13 . . . 231
8.1212 so_7084831_7085249_10 . . . 231
8.1213 so_16838591_16838644_8 . . . 231
8.1214 so_16838591_16838644_9 . . . 231
8.1215 so_16838591_16838644_6 . . . 231
8.1216 so_16838591_16838644_7 . . . 231
8.1217 so_8943261_8943287_9 . . . 231
8.1218 so_8943261_8943287_6 . . . 231
8.1219 so_6284507_6284695_1 . . . 231
8.1220 so_38131945_38131963_4 . . . 231
8.1221 so_27382036_27388378_5 . . . 231
8.1222 so_27382036_27388378_4 . . . 231
8.1223 so_27382036_27388378_3 . . . 231
8.1224 so_47508477_47508870_0 . . . 231
8.1225 so_23987515_23987926_3 . . . 231
8.1226 so_23987515_23987926_6 . . . 231
8.1227 so_23987515_23987926_5 . . . 231
8.1228 so_5560363_5560415_0 . . . 231
8.1229 so_28969309_28969576_2 . . . 231
8.1230 so_29765961_29767272_5 . . . 231
8.1231 so_29765961_29766171_3 . . . 231

8.1232 so_29765961_29766171_4 . . . 231
8.1233 so_14895428_14895493_5 . . . 231
8.1234 so_14895428_14895493_4 . . . 231
8.1235 so_14895428_14895493_2 . . . 231
8.1236 so_14895428_14895494_2 . . . 231
8.1237 so_14895428_14895494_1 . . . 231
8.1238 so_12164061_12164116_2 . . . 231
8.1239 so_12164061_12164116_5 . . . 231
8.1240 so_12164061_12164116_4 . . . 231
8.1241 so_11058422_11058463_7 . . . 231
8.1242 so_11058422_11058463_6 . . . 231
8.1243 so_11058422_11058463_4 . . . 231
8.1244 so_11058422_11058463_5 . . . 231
8.1245 so_33913572_33913851_2 . . . 231
8.1246 so_5602030_5602057_1 . . . 231
8.1247 so_13423007_13423263_2 . . . 231
8.1248 so_28391796_28391896_1 . . . 231
8.1249 so_32377780_32377854_5 . . . 231
8.1250 so_32377780_32377854_4 . . . 231
8.1251 so_21023071_21023071_0 . . . 231
8.1252 so_21023071_21023071_1 . . . 231
8.1253 so_21023071_21023071_2 . . . 231
8.1254 so_21023071_21023071_3 . . . 231
8.1255 so_27200284_27200678_3 . . . 231
8.1256 so_27200284_27200678_4 . . . 231
8.1257 so_14259602_14259765_2 . . . 231
8.1258 so_27484483_27484598_3 . . . 231
8.1259 so_18295302_18304935_0 . . . 231
8.1260 so_18295302_18304935_1 . . . 231
8.1261 so_23334719_23335067_13 . . . 231
8.1262 so_23334719_23335067_10 . . . 231
8.1263 so_23334719_23335294_1 . . . 231
8.1264 so_23334719_23335067_12 . . . 231
8.1265 so_12580171_12580210_2 . . . 231
8.1266 so_39577922_39577922_2 . . . 231
8.1267 so_39577922_39577922_9 . . . 231
8.1268 so_39577922_39577922_10 . . . 231
8.1269 so_32648805_32652158_3 . . . 231
8.1270 so_32783394_32783472_5 . . . 231
8.1271 so_27668269_27668327_3 . . . 231
8.1272 so_27668269_27668438_3 . . . 231
8.1273 so_27668269_27668327_2 . . . 231
8.1274 so_27668269_27668438_4 . . . 231
8.1275 so_26699720_26700151_6 . . . 231
8.1276 so_26699720_26699720_2 . . . 231
8.1277 so_26699720_26699720_1 . . . 231
8.1278 so_26699720_26700151_8 . . . 231
8.1279 so_26699720_26700151_7 . . . 231
8.1280 so_26699720_26699720_3 . . . 231
8.1281 so_5610352_5610375_2 . . . 231
8.1282 so_17830690_17831711_2 . . . 231

8.1283 so_20849454_20849500_2 . . . 231
8.1284 so_20849454_20849500_4 . . . 231
8.1285 so_20849454_20849500_5 . . . 231
8.1286 so_13576055_23698999_2 . . . 231
8.1287 so_13576055_23698999_0 . . . 231
8.1288 so_13576055_23698999_3 . . . 231
8.1289 so_27699118_27699260_5 . . . 231
8.1290 so_27699118_27699260_2 . . . 231
8.1291 so_27699118_27699260_3 . . . 231
8.1292 so_29291371_29292092_6 . . . 231
8.1293 so_29291371_29292092_5 . . . 231
8.1294 so_4539563_4539588_1 . . . 231
8.1295 so_3580789_3580876_4 . . . 231
8.1296 so_3580789_3580876_1 . . . 231
8.1297 so_1628311_1628396_4 . . . 231
8.1298 so_23882409_23882409_0 . . . 231
8.1299 so_39429510_39429696_2 . . . 231
8.1300 so_17257660_17257681_8 . . . 231
8.1301 so_17257660_17257681_9 . . . 231
8.1302 so_17257660_17257681_6 . . . 231
8.1303 so_42534992_42535090_1 . . . 231
8.1304 so_6511971_6512058_1 . . . 231
8.1305 so_25394665_25394763_1 . . . 231
8.1306 so_9342758_9342993_13 . . . 231
8.1307 so_9342758_9342993_11 . . . 231
8.1308 so_9342758_9342993_14 . . . 231
8.1309 so_9342758_9342993_16 . . . 231
8.1310 so_22265255_22265499_8 . . . 231
8.1311 so_22265255_22265499_0 . . . 231
8.1312 so_22265255_22265499_7 . . . 231
8.1313 so_40098080_40098140_2 . . . 231
8.1314 so_1613341_1613383_2 . . . 231
8.1315 so_14829158_14829622_3 . . . 231
8.1316 so_14829158_14829622_5 . . . 231
8.1317 so_14829158_14829622_6 . . . 231
8.1318 so_13904020_13913553_0 . . . 231
8.1319 so_15548813_15548873_2 . . . 231
8.1320 so_15548813_15548873_4 . . . 231
8.1321 so_15548813_15548855_1 . . . 231
8.1322 so_15548813_15548873_3 . . . 231
8.1323 so_15548813_15548978_2 . . . 231
8.1324 so_15548813_15548873_5 . . . 231
8.1325 so_13097781_13097841_2 . . . 231
8.1326 so_29683381_29683772_3 . . . 231
8.1327 so_10828294_10828333_6 . . . 231
8.1328 so_25198405_25198802_3 . . . 231
8.1329 so_2836939_2836939_6 . . . 231
8.1330 so_19121890_19122919_7 . . . 231
8.1331 so_24043897_24044147_0 . . . 231
8.1332 so_24043897_24043897_1 . . . 231
8.1333 so_24043897_24044147_1 . . . 231

8.1334 so_24043897_24044147_2 . . . 231
8.1335 so_17257828_17257838_6 . . . 231
8.1336 so_17257828_17257838_8 . . . 231
8.1337 so_17257828_17257838_9 . . . 231
8.1338 so_22849688_22849688_14 . . . 231
8.1339 so_22849688_22849688_13 . . . 231
8.1340 so_22849688_22849688_11 . . . 231
8.1341 so_15092931_15093382_1 . . . 231
8.1342 so_9025792_9025981_2 . . . 231
8.1343 so_22233148_22234327_6 . . . 231
8.1344 so_27349679_27349679_4 . . . 231
8.1345 so_27349679_27349771_3 . . . 231
8.1346 so_27349679_27349679_7 . . . 231
8.1347 so_15080742_15080926_9 . . . 231
8.1348 so_15080742_15080926_6 . . . 231
8.1349 so_15080742_15080926_8 . . . 231
8.1350 so_16339811_16340318_4 . . . 231
8.1351 so_18901315_18902462_8 . . . 231
8.1352 so_35383453_35383576_13 . . . 231
8.1353 so_20936574_20937337_1 . . . 231
8.1354 so_20936574_20936574_1 . . . 231
8.1355 so_20936574_20936695_3 . . . 231
8.1356 so_20936574_20937337_0 . . . 231
8.1357 so_20936574_20936574_0 . . . 231
8.1358 so_5950144_5950176_1 . . . 231
8.1359 so_11993134_11993284_2 . . . 231
8.1360 so_1751378_1751546_4 . . . 231
8.1361 so_1751378_1751546_1 . . . 231
8.1362 so_4523959_12965621_5 . . . 231
8.1363 so_26947704_26949366_3 . . . 231
8.1364 so_28717682_28718438_3 . . . 232
8.1365 so_15149363_15149482_3 . . . 232
8.1366 so_5474349_5474523_0 . . . 232
8.1367 so_5474349_5474446_1 . . . 232
8.1368 so_16300432_16302929_1 . . . 232
8.1369 so_6189096_6189122_3 . . . 232
8.1370 so_4378785_4379221_0 . . . 232
8.1371 so_31389454_31389824_1 . . . 232
8.1372 so_14774152_14774430_4 . . . 232
8.1373 so_14774152_14774430_0 . . . 232
8.1374 so_14774152_14774430_3 . . . 232
8.1375 so_19658609_19658743_0 . . . 232
8.1376 so_1628311_1628396_5 . . . 232
8.1377 so_21350478_21350570_0 . . . 232
8.1378 so_8400791_8400791_4 . . . 232
8.1379 so_17104209_17104608_5 . . . 232
8.1380 so_2318650_2318731_0 . . . 232
8.1381 so_2318650_2318731_2 . . . 232
8.1382 so_27642882_27643207_0 . . . 232
8.1383 so_11812687_11812718_5 . . . 232
8.1384 so_19575202_19575287_1 . . . 232

8.1385 so_25903968_25904184_8 . . . 232
8.1386 so_11164394_11164416_0 . . . 232
8.1387 so_27090093_27090513_0 . . . 232
8.1388 so_27090093_27090513_1 . . . 232
8.1389 so_8697105_8697182_3 . . . 232
8.1390 so_23987515_23987926_1 . . . 232
8.1391 so_5560363_5560415_2 . . . 232
8.1392 so_25964198_25964198_0 . . . 232
8.1393 so_29765961_29766171_2 . . . 232
8.1394 so_4121296_4121323_0 . . . 232
8.1395 so_12364074_12629563_1 . . . 232
8.1396 so_11058422_11058463_10 . . . 232
8.1397 so_5602030_5602057_0 . . . 232
8.1398 so_32377780_32377854_3 . . . 232
8.1399 so_14259602_14259765_1 . . . 232
8.1400 so_14259602_14259760_0 . . . 232
8.1401 so_27484483_27484598_2 . . . 232
8.1402 so_32032249_32034107_1 . . . 232
8.1403 so_23334719_23335294_0 . . . 232
8.1404 so_23334719_23335067_8 . . . 232
8.1405 so_34856965_34857406_0 . . . 232
8.1406 so_26120395_26120507_3 . . . 232
8.1407 so_26120395_26120507_0 . . . 232
8.1408 so_26120395_26120507_1 . . . 232
8.1409 so_27668269_27668327_1 . . . 232
8.1410 so_17830690_17831711_3 . . . 232
8.1411 so_17830690_17831711_5 . . . 232
8.1412 so_17830690_17831711_1 . . . 232
8.1413 so_27699118_27699260_0 . . . 232
8.1414 so_1628311_1628396_6 . . . 232
8.1415 so_4296090_4296195_0 . . . 232
8.1416 so_4296090_4296195_3 . . . 232
8.1417 so_6511971_6512058_0 . . . 232
8.1418 so_22265255_22265499_6 . . . 232
8.1419 so_40098080_40098140_1 . . . 232
8.1420 so_17713124_17713134_1 . . . 232
8.1421 so_17713124_17713134_0 . . . 232
8.1422 so_14829158_14829622_1 . . . 232
8.1423 so_15548813_15548873_1 . . . 232
8.1424 so_15548813_15548855_0 . . . 232
8.1425 so_180172_31093033_0 . . . 232
8.1426 so_14533172_14533216_0 . . . 232
8.1427 so_14533172_14533216_3 . . . 232
8.1428 so_4950125_4950186_5 . . . 232
8.1429 so_19121890_19122919_11 . . . 232
8.1430 so_15092931_15093382_0 . . . 232
8.1431 so_27349679_27349771_1 . . . 232
8.1432 so_15080742_15080926_4 . . . 232
8.1433 so_5950144_5950176_0 . . . 232
8.1434 so_4523959_12965621_3 . . . 232
8.1435 so_4294100_4294152_3 . . . 232

8.1436 so_4294100_4294152_0 . . . 232
8.1437 so_32072029_32072151_2 . . . 232
8.1438 so_41890635_41902202_1 . . . 232
8.1439 so_35449320_35449495_3 . . . 232
8.1440 so_33126333_33126532_0 . . . 232
8.1441 so_25438023_25438102_2 . . . 232
8.1442 so_15149363_15149482_0 . . . 232
8.1443 so_27421419_27421507_0 . . . 232
8.1444 so_5498619_46071432_0 . . . 232
8.1445 so_20008939_20008969_2 . . . 232
8.1446 so_17106389_17106428_1 . . . 232
8.1447 so_4285895_30700051_0 . . . 232
8.1448 so_4285895_30700051_1 . . . 232
8.1449 so_17276416_17276482_1 . . . 232
8.1450 so_10359289_10359328_2 . . . 232
8.1451 so_11508466_11508524_0 . . . 232
8.1452 so_26077807_26077807_4 . . . 232
8.1453 so_45506400_45506783_0 . . . 232
8.1454 so_45506400_45507183_7 . . . 232
8.1455 so_45506400_45507183_4 . . . 232
8.1456 so_45506400_45507183_8 . . . 232
8.1457 so_45506400_45507183_1 . . . 232
8.1458 so_45506400_45507183_6 . . . 232
8.1459 so_45506400_45507183_2 . . . 232
8.1460 so_45506400_45506714_2 . . . 232
8.1461 so_29525797_29528757_3 . . . 232
8.1462 so_28001512_28001552_0 . . . 232
8.1463 so_24338968_24339068_0 . . . 232
8.1464 so_3437110_3437197_3 . . . 232
8.1465 so_3437110_3437197_5 . . . 232
8.1466 so_27382036_27388378_0 . . . 232
8.1467 so_1895922_1898691_2 . . . 232
8.1468 so_25964198_25964198_1 . . . 232
8.1469 so_15030379_15030445_0 . . . 232
8.1470 so_29109654_29109746_2 . . . 232
8.1471 so_32377780_32377854_0 . . . 232
8.1472 so_30596984_30597081_0 . . . 232
8.1473 so_31389454_31391081_2 . . . 232
8.1474 so_27484483_27484598_0 . . . 232
8.1475 so_30180417_30180440_9 . . . 232
8.1476 so_30180417_30180440_12 . . . 232
8.1477 so_30180417_30180440_2 . . . 232
8.1478 so_30180417_30180440_10 . . . 232
8.1479 so_30180417_30180691_1 . . . 232
8.1480 so_30180417_30180691_0 . . . 232
8.1481 so_30180417_30180440_5 . . . 232
8.1482 so_30180417_30180440_3 . . . 232
8.1483 so_30180417_30180440_11 . . . 232
8.1484 so_30180417_30180440_4 . . . 232
8.1485 so_30180417_30180440_1 . . . 232
8.1486 so_23334719_23335067_1 . . . 232

8.1487 so_5813700_5813736_0 . . . 232
8.1488 so_47437349_47440920_20 . . . 232
8.1489 so_47437349_47440920_16 . . . 232
8.1490 so_47437349_47440920_15 . . . 232
8.1491 so_47437349_47440920_18 . . . 232
8.1492 so_47437349_47440920_12 . . . 232
8.1493 so_47437349_47440920_14 . . . 232
8.1494 so_47437349_47440920_19 . . . 232
8.1495 so_47437349_47440920_13 . . . 232
8.1496 so_21964120_21964142_2 . . . 232
8.1497 so_25999309_25999455_4 . . . 232
8.1498 so_25999309_25999448_0 . . . 232
8.1499 so_17830690_17831711_0 . . . 232
8.1500 so_11153863_11153923_1 . . . 232
8.1501 so_27595793_27595998_1 . . . 232
8.1502 so_27595793_27595993_0 . . . 232
8.1503 so_27699118_27699462_4 . . . 232
8.1504 so_27699118_27699462_3 . . . 232
8.1505 so_27699118_27699260_16 . . . 232
8.1506 so_40943080_40943130_0 . . . 232
8.1507 so_41596068_41596068_0 . . . 232
8.1508 so_22795037_22795079_0 . . . 232
8.1509 so_33456141_33457935_0 . . . 232
8.1510 so_33456141_33457935_1 . . . 232
8.1511 so_25395297_25395408_5 . . . 232
8.1512 so_31502894_31502943_0 . . . 232
8.1513 so_29132375_29133610_0 . . . 232
8.1514 so_42928235_42928291_2 . . . 232
8.1515 so_42928235_42928291_4 . . . 232
8.1516 so_42928235_42928291_3 . . . 232
8.1517 so_33585094_33585174_0 . . . 232
8.1518 so_23415661_23415662_0 . . . 232
8.1519 so_23415661_23415662_2 . . . 232
8.1520 so_23415661_23415662_1 . . . 232
8.1521 so_23415661_23415662_3 . . . 232
8.1522 so_23415661_23415662_5 . . . 232
8.1523 so_23415661_23415662_8 . . . 232
8.1524 so_23415661_23415662_9 . . . 232
8.1525 so_23415661_23415662_7 . . . 232
8.1526 so_22265255_22265499_1 . . . 232
8.1527 so_26153165_26153187_0 . . . 232
8.1528 so_8092670_8092791_1 . . . 232
8.1529 so_14829158_14829622_17 . . . 232
8.1530 so_29868792_29873914_7 . . . 232
8.1531 so_24651447_24651710_1 . . . 232
8.1532 so_29683381_29683772_0 . . . 232
8.1533 so_39742393_39742701_0 . . . 232
8.1534 so_1119370_1119390_5 . . . 232
8.1535 so_2836939_2836939_9 . . . 232
8.1536 so_4279264_23429477_0 . . . 232
8.1537 so_2890598_2890604_0 . . . 232

8.1538 so_17257828_17257838_1 . . . 232
8.1539 so_22233148_23955849_1 . . . 232
8.1540 so_45429780_45429894_3 . . . 232
8.1541 so_45429780_45429894_4 . . . 232
8.1542 so_35383453_35383576_17 . . . 232
8.1543 so_33398579_33398853_8 . . . 232
8.1544 so_11993134_11993284_0 . . . 232
8.1545 so_14659752_14659931_7 . . . 232
8.1546 so_14792068_14792098_0 . . . 232
8.1547 so_47311250_47314810_0 . . . 232
8.1548 so_21468642_21469852_2 . . . 232
8.1549 so_24293311_24313401_7 . . . 233
8.1550 so_27698056_27698576_0 . . . 233
8.1551 so_1816547_1816578_1 . . . 233
8.1552 so_24500164_24500672_0 . . . 233
8.1553 so_5350125_5350209_0 . . . 233
8.1554 so_21150067_21169201_0 . . . 233
8.1555 so_5064296_5064470_1 . . . 233
8.1556 so_5064296_5064937_0 . . . 233
8.1557 so_13952918_13953061_0 . . . 233
8.1558 so_11069605_11069605_0 . . . 233
8.1559 so_3249147_3249392_1 . . . 233
8.1560 so_25822574_25822612_0 . . . 233
8.1561 so_26204671_26204671_0 . . . 233
8.1562 so_7457370_7457436_0 . . . 233
8.1563 so_6358073_6358511_0 . . . 233
8.1564 so_17770935_17773924_3 . . . 233
8.1565 so_33832423_33832712_1 . . . 233
8.1566 so_25795442_25795685_1 . . . 233
8.1567 so_43432185_43432270_0 . . . 233
8.1568 so_14912800_14913148_2 . . . 233
8.1569 so_25832908_25832969_1 . . . 233
8.1570 so_37254813_37254866_3 . . . 233
8.1571 so_26244887_26245704_3 . . . 233
8.1572 so_26244887_26245704_5 . . . 233
8.1573 so_13693871_13693871_2 . . . 233
8.1574 so_18625172_18625172_0 . . . 233
8.1575 so_17218769_17218854_1 . . . 233
8.1576 so_11069605_11069605_1 . . . 233
8.1577 so_3249147_3249392_2 . . . 233
8.1578 so_3848429_3848461_2 . . . 233
8.1579 so_14967167_14967452_2 . . . 233
8.1580 so_15077466_15077788_3 . . . 233
8.1581 so_25534628_25534823_4 . . . 233
8.1582 so_4153527_4154213_1 . . . 233
8.1583 so_32648805_32652158_0 . . . 233
8.1584 so_2027873_2027988_0 . . . 233
8.1585 so_7457370_7457436_2 . . . 233
8.1586 so_39127703_39127999_1 . . . 233
8.1587 so_7006202_7006249_1 . . . 233
8.1588 so_6358073_6358511_2 . . . 233

8.1589 so_26612554_26612762_1 . . . 233
8.1590 so_25607216_25609753_1 . . . 233
8.1591 so_4285608_4285641_0 . . . 233
8.1592 so_41594186_41707564_0 . . . 233
8.1593 so_29834308_29834426_1 . . . 233
8.1594 so_21708606_21710850_1 . . . 233
8.1595 so_31572915_31574338_0 . . . 233
8.1596 so_27352021_27354688_5 . . . 233
8.1597 so_2838107_2838701_5 . . . 233
8.1598 so_32415552_32415678_0 . . . 233
8.1599 so_14543554_14543600_0 . . . 233
8.1600 so_15286450_15286450_2 . . . 233
8.1601 so_25795442_25795685_4 . . . 233
8.1602 so_24720028_24720311_2 . . . 233
8.1603 so_25832908_25832969_3 . . . 233
8.1604 so_39109710_39109943_0 . . . 233
8.1605 so_13693871_13693871_6 . . . 233
8.1606 so_20973417_20973417_1 . . . 233
8.1607 so_5064296_5064937_2 . . . 233
8.1608 so_10843715_10843772_0 . . . 233
8.1609 so_25964198_25964198_2 . . . 233
8.1610 so_3249147_3249392_3 . . . 233
8.1611 so_38065676_39728792_2 . . . 233
8.1612 so_4153527_4154213_2 . . . 233
8.1613 so_32783394_32783472_2 . . . 233
8.1614 so_2027873_2027988_1 . . . 233
8.1615 so_22603979_22604146_6 . . . 233
8.1616 so_22603979_22604146_9 . . . 233
8.1617 so_6358073_6358511_4 . . . 233
8.1618 so_30012733_30013834_3 . . . 233
8.1619 so_35383453_35383576_11 . . . 233
8.1620 so_38935256_38966560_1 . . . 233
8.1621 so_11465999_11472302_1 . . . 233
8.1622 so_31896298_31897188_0 . . . 233
8.1623 so_8464400_8585258_0 . . . 233
8.1624 so_43432185_43432270_4 . . . 233
8.1625 so_43432185_43432270_3 . . . 233
8.1626 so_43432185_43432270_5 . . . 233
8.1627 so_45698374_45772026_1 . . . 233
8.1628 so_17104209_17104573_2 . . . 233
8.1629 so_10614728_10614894_1 . . . 233
8.1630 so_44151330_44151610_4 . . . 233
8.1631 so_44151330_44151610_1 . . . 233
8.1632 so_44151330_44151610_6 . . . 233
8.1633 so_44151330_44151610_3 . . . 233
8.1634 so_44151330_44151610_2 . . . 233
8.1635 so_18089988_18089988_1 . . . 233
8.1636 so_18089988_18089988_3 . . . 233
8.1637 so_18089988_18090865_0 . . . 233
8.1638 so_4283307_4283362_1 . . . 233
8.1639 so_4283307_4283362_0 . . . 233

8.1640 so_22661317_22662143_0 . . . 233
8.1641 so_18538427_18538427_1 . . . 233
8.1642 so_13459057_13460468_2 . . . 233
8.1643 so_13459057_13460468_0 . . . 233
8.1644 so_39109710_39140477_3 . . . 233
8.1645 so_42276302_42276668_1 . . . 233
8.1646 so_1445274_1445327_3 . . . 233
8.1647 so_24678701_24679050_1 . . . 233
8.1648 so_37254813_37254866_6 . . . 233
8.1649 so_14935722_14991772_4 . . . 233
8.1650 so_7401521_7401652_0 . . . 233
8.1651 so_32521974_32522401_0 . . . 233
8.1652 so_46162766_46163082_2 . . . 233
8.1653 so_1384007_1384044_1 . . . 233
8.1654 so_24500164_24500918_2 . . . 233
8.1655 so_24500164_24500918_0 . . . 233
8.1656 so_24500164_24500918_4 . . . 233
8.1657 so_24500164_24500918_1 . . . 233
8.1658 so_47546111_47546260_1 . . . 233
8.1659 so_13693871_13694435_0 . . . 233
8.1660 so_18625172_18625172_1 . . . 233
8.1661 so_27249698_27249698_0 . . . 233
8.1662 so_38535292_38535667_3 . . . 233
8.1663 so_38535292_38535667_2 . . . 233
8.1664 so_27382036_27388378_7 . . . 233
8.1665 so_27752546_27757519_0 . . . 233
8.1666 so_27645671_27645680_0 . . . 233
8.1667 so_44311306_44311825_2 . . . 233
8.1668 so_44311306_44311825_1 . . . 233
8.1669 so_44311306_44311825_0 . . . 233
8.1670 so_20973417_20973417_9 . . . 233
8.1671 so_20973417_20973417_7 . . . 233
8.1672 so_20973417_20973417_4 . . . 233
8.1673 so_20973417_21044373_0 . . . 233
8.1674 so_20973417_20973417_3 . . . 233
8.1675 so_20973417_20973417_5 . . . 233
8.1676 so_20973417_20973417_11 . . . 233
8.1677 so_20973417_20973417_10 . . . 233
8.1678 so_20973417_20973417_2 . . . 233
8.1679 so_20973417_20973417_8 . . . 233
8.1680 so_20973417_20973417_12 . . . 233
8.1681 so_15412762_15412927_0 . . . 233
8.1682 so_26788402_26788564_0 . . . 233
8.1683 so_5064296_5064937_3 . . . 233
8.1684 so_5064296_5064470_0 . . . 233
8.1685 so_44853774_44853774_0 . . . 233
8.1686 so_44853774_44853864_0 . . . 233
8.1687 so_41348082_41348082_10 . . . 233
8.1688 so_33970892_33978450_2 . . . 233
8.1689 so_32835803_32835960_2 . . . 233
8.1690 so_32835803_32835960_3 . . . 233

8.1691 so_32835803_32835960_0 . . . 233
8.1692 so_32835803_32835960_5 . . . 233
8.1693 so_32835803_32835960_1 . . . 233
8.1694 so_25964198_25964337_0 . . . 233
8.1695 so_25964198_25964198_3 . . . 233
8.1696 so_26685551_26963841_0 . . . 233
8.1697 so_40018076_40019058_1 . . . 233
8.1698 so_3021333_3021396_1 . . . 233
8.1699 so_27205927_27205927_0 . . . 233
8.1700 so_38065676_39728792_3 . . . 233
8.1701 so_14712802_14713272_1 . . . 233
8.1702 so_14712802_14713272_3 . . . 233
8.1703 so_14712802_14713272_0 . . . 233
8.1704 so_6376580_6376875_2 . . . 233
8.1705 so_6376580_6376743_1 . . . 233
8.1706 so_14967167_14967452_3 . . . 233
8.1707 so_32377780_32377854_2 . . . 233
8.1708 so_27314485_27314556_4 . . . 233
8.1709 so_27314485_27314556_1 . . . 233
8.1710 so_27314485_27314556_3 . . . 233
8.1711 so_40533815_40534093_2 . . . 233
8.1712 so_31550990_31552110_0 . . . 233
8.1713 so_353632_353649_0 . . . 233
8.1714 so_30180417_30180440_8 . . . 233
8.1715 so_20455382_20455655_0 . . . 233
8.1716 so_45551177_45804869_1 . . . 233
8.1717 so_45551177_45804869_2 . . . 233
8.1718 so_45551177_45804869_3 . . . 233
8.1719 so_45551177_45804869_0 . . . 233
8.1720 so_45551177_45804869_5 . . . 233
8.1721 so_31165342_31165342_1 . . . 233
8.1722 so_40006925_40010380_2 . . . 233
8.1723 so_33074102_33074102_7 . . . 233
8.1724 so_33074102_33074102_9 . . . 233
8.1725 so_33074102_33074102_8 . . . 233
8.1726 so_33074102_33074102_11 . . . 233
8.1727 so_33074102_33074102_10 . . . 233
8.1728 so_3546340_3546582_3 . . . 233
8.1729 so_46909105_46909166_1 . . . 233
8.1730 so_39971659_39971950_0 . . . 233
8.1731 so_39971659_39971691_1 . . . 233
8.1732 so_39971659_39971950_1 . . . 233
8.1733 so_27303650_27303980_4 . . . 233
8.1734 so_27303650_27304057_0 . . . 233
8.1735 so_32783394_32783472_3 . . . 233
8.1736 so_17676440_17676499_2 . . . 233
8.1737 so_17676440_17676499_3 . . . 233
8.1738 so_33242948_33242948_0 . . . 233
8.1739 so_33242948_33243400_0 . . . 233
8.1740 so_46071992_46072220_0 . . . 233
8.1741 so_28133607_28133607_0 . . . 233

8.1742 so_28133607_28133607_2 . . . 233
8.1743 so_28457832_28458512_1 . . . 233
8.1744 so_12847272_12848540_0 . . . 233
8.1745 so_12847272_12848540_1 . . . 233
8.1746 so_12847272_12848540_2 . . . 233
8.1747 so_7457370_7457436_3 . . . 233
8.1748 so_27699118_27699260_12 . . . 233
8.1749 so_27699118_27699260_9 . . . 233
8.1750 so_21825933_21836197_4 . . . 233
8.1751 so_21825933_21836197_2 . . . 233
8.1752 so_21825933_21836197_11 . . . 233
8.1753 so_21825933_21836197_8 . . . 233
8.1754 so_21825933_21836197_3 . . . 233
8.1755 so_21825933_21836197_10 . . . 233
8.1756 so_21825933_21836197_7 . . . 233
8.1757 so_21825933_21836197_6 . . . 233
8.1758 so_21825933_21836197_9 . . . 233
8.1759 so_29723346_29723627_0 . . . 233
8.1760 so_42727765_42728073_1 . . . 233
8.1761 so_31728686_31729647_0 . . . 233
8.1762 so_23882409_23882409_1 . . . 233
8.1763 so_37701321_37707777_3 . . . 233
8.1764 so_44015848_44016076_1 . . . 233
8.1765 so_44015848_44016076_0 . . . 233
8.1766 so_25589674_25589764_0 . . . 233
8.1767 so_11222076_11222105_1 . . . 233
8.1768 so_29931718_29931857_0 . . . 233
8.1769 so_7006202_7006249_2 . . . 233
8.1770 so_6358073_6358511_6 . . . 233
8.1771 so_6358073_6358511_5 . . . 233
8.1772 so_38801955_38801955_0 . . . 233
8.1773 so_25395056_25395056_1 . . . 233
8.1774 so_35962647_35962759_0 . . . 233
8.1775 so_7979855_7980055_0 . . . 233
8.1776 so_18854404_18854682_0 . . . 233
8.1777 so_22243721_22243785_6 . . . 233
8.1778 so_27845061_27845171_1 . . . 233
8.1779 so_11911056_11911111_1 . . . 233
8.1780 so_44306607_44310738_1 . . . 233
8.1781 so_44306607_44310738_2 . . . 233
8.1782 so_25745759_25758367_1 . . . 233
8.1783 so_32270244_32270347_0 . . . 233
8.1784 so_32053359_32053869_0 . . . 233
8.1785 so_20850536_20999389_9 . . . 233
8.1786 so_27182198_27182198_0 . . . 233
8.1787 so_27182198_27182198_2 . . . 233
8.1788 so_27182198_27182198_1 . . . 233
8.1789 so_8535226_8535257_3 . . . 233
8.1790 so_21481462_21481722_0 . . . 233
8.1791 so_46427709_46430666_1 . . . 233
8.1792 so_21822438_21822438_0 . . . 233

8.1793 so_21822438_21822438_2 . . . 233
8.1794 so_21822438_21822438_1 . . . 233
8.1795 so_41594186_41707564_8 . . . 233
8.1796 so_41594186_41707564_1 . . . 233
8.1797 so_21708606_21710850_4 . . . 233
8.1798 so_21708606_21710850_2 . . . 233
8.1799 so_21708606_21710850_3 . . . 233
8.1800 so_32427433_32428111_1 . . . 233
8.1801 so_35383453_35383576_12 . . . 233
8.1802 so_25360835_25362435_0 . . . 233
8.1803 so_25360835_25362435_2 . . . 233
8.1804 so_25360835_25362435_1 . . . 233
8.1805 so_34767949_34767949_0 . . . 233
8.1806 so_31572915_31574338_1 . . . 233
8.1807 so_4801136_4801136_0 . . . 233
8.1808 so_25241516_25242107_0 . . . 233
8.1809 so_25241516_25242107_3 . . . 233
8.1810 so_25241516_25242107_2 . . . 233
8.1811 so_10048130_10048295_1 . . . 233
8.1812 so_1758142_9945598_0 . . . 233
8.1813 so_33832423_33832712_0 . . . 233
8.1814 so_11223285_11223647_2 . . . 233
8.1815 so_11223285_11223647_1 . . . 233
8.1816 so_4300869_4300911_1 . . . 233
8.1817 so_38487640_38488280_2 . . . 233
8.1818 so_37446947_37447085_0 . . . 234
8.1819 so_14016993_14017039_3 . . . 234
8.1820 so_26876310_26876310_1 . . . 234
8.1821 so_18155239_18155370_2 . . . 234
8.1822 so_23248505_23248505_0 . . . 234
8.1823 so_7254698_7257081_1 . . . 234
8.1824 so_26920134_26920239_1 . . . 234
8.1825 so_28408911_28409778_2 . . . 234
8.1826 so_20576162_20576364_1 . . . 234
8.1827 so_29571705_29571782_0 . . . 234
8.1828 so_16983539_16983926_0 . . . 234
8.1829 so_19112399_19112572_1 . . . 234
8.1830 so_11984297_11984594_0 . . . 234
8.1831 so_5442717_5442736_0 . . . 234
8.1832 so_23188173_23188203_0 . . . 234
8.1833 so_4178175_27511360_0 . . . 234
8.1834 so_4178175_4178176_0 . . . 234
8.1835 so_4178175_7189821_0 . . . 234
8.1836 so_18581443_18581492_2 . . . 234
8.1837 so_17457579_19258426_0 . . . 234
8.1838 so_17294925_17299103_0 . . . 234
8.1839 so_29989250_29989381_0 . . . 234
8.1840 so_43694112_43694276_0 . . . 234
8.1841 so_7215647_7372003_0 . . . 234
8.1842 so_8758445_8758760_0 . . . 234
8.1843 so_29399870_29401520_1 . . . 234

8.1844 so_29399870_29401520_0 . . . 234
8.1845 so_25300251_25300251_5 . . . 234
8.1846 so_12162893_12163119_1 . . . 234
8.1847 so_17232890_17233157_0 . . . 234
8.1848 so_17232890_17233157_1 . . . 234
8.1849 so_33988297_33989033_0 . . . 234
8.1850 so_35891233_35891969_1 . . . 234
8.1851 so_35891233_35891969_0 . . . 234
8.1852 so_2128307_2128328_0 . . . 234
8.1853 so_13886101_13886302_0 . . . 234
8.1854 so_38065676_39728792_0 . . . 234
8.1855 so_13423007_13423263_0 . . . 234
8.1856 so_28677593_28685181_3 . . . 234
8.1857 so_33553980_33556166_1 . . . 234
8.1858 so_33553980_33556166_0 . . . 234
8.1859 so_12911527_12911541_0 . . . 234
8.1860 so_28859452_28859535_0 . . . 234
8.1861 so_34231132_34231167_0 . . . 234
8.1862 so_39267963_39273340_1 . . . 234
8.1863 so_24043577_24043750_1 . . . 234
8.1864 so_24043577_24043750_0 . . . 234
8.1865 so_27118535_27118551_1 . . . 234
8.1866 so_27118535_27118551_0 . . . 234
8.1867 so_24958126_24958839_0 . . . 234
8.1868 so_3952020_3952053_0 . . . 234
8.1869 so_16606841_16607656_0 . . . 234
8.1870 so_27668269_27668327_0 . . . 234
8.1871 so_45834990_45835907_4 . . . 234
8.1872 so_45834990_45835907_6 . . . 234
8.1873 so_45834990_45835907_1 . . . 234
8.1874 so_11742592_11742888_1 . . . 234
8.1875 so_32940847_32946270_0 . . . 234
8.1876 so_146452_146589_2 . . . 234
8.1877 so_37618213_37618395_1 . . . 234
8.1878 so_33522985_33523669_2 . . . 234
8.1879 so_23882409_23883035_17 . . . 234
8.1880 so_39429510_39431986_2 . . . 234
8.1881 so_29970221_29970349_1 . . . 234
8.1882 so_39376813_39379359_3 . . . 234
8.1883 so_39376813_39379359_5 . . . 234
8.1884 so_23892018_23965233_1 . . . 234
8.1885 so_23892018_23965233_2 . . . 234
8.1886 so_30077670_30083769_0 . . . 234
8.1887 so_2792028_2792103_1 . . . 234
8.1888 so_5666321_5666342_1 . . . 234
8.1889 so_3674247_3679872_0 . . . 234
8.1890 so_3674247_3674759_1 . . . 234
8.1891 so_6703288_6703484_0 . . . 234
8.1892 so_23808357_23809231_0 . . . 234
8.1893 so_34856936_34857037_0 . . . 234
8.1894 so_26193718_26193854_1 . . . 234

8.1895 so_26193718_26193854_0 . . . 234
8.1896 so_47096920_47097250_0 . . . 234
8.1897 so_13625774_13626075_0 . . . 234
8.1898 so_21340801_21341177_0 . . . 234
8.1899 so_12596158_12596333_0 . . . 234
8.1900 so_88957_88960_0 . . . 234
8.1901 so_18216313_18216727_0 . . . 234
8.1902 so_31793983_31794110_0 . . . 234
8.1903 so_23304025_23304090_2 . . . 234
8.1904 so_7189813_7189848_0 . . . 234
8.1905 so_16595715_16595788_0 . . . 234
8.1906 so_25173462_25173547_3 . . . 234
8.1907 so_3454954_3455012_1 . . . 234
8.1908 so_13551572_13551661_3 . . . 234
8.1909 so_19590008_19590429_2 . . . 234
8.1910 so_18901315_18902462_0 . . . 234
8.1911 so_44533602_44533715_5 . . . 234
8.1912 so_44533602_44533715_3 . . . 234
8.1913 so_44533602_44533715_0 . . . 234
8.1914 so_33297916_33298199_1 . . . 234
8.1915 so_33297916_33298199_2 . . . 234
8.1916 so_20641735_20963668_0 . . . 234
8.1917 so_24372236_24372309_0 . . . 234
8.1918 so_36454463_36454463_1 . . . 234
8.1919 so_40238484_40238583_3 . . . 234
8.1920 so_29095153_29095435_0 . . . 234
8.1921 so_2838107_2838701_2 . . . 234
8.1922 so_27521145_27521145_1 . . . 234
8.1923 so_11394924_11394947_0 . . . 234
8.1924 so_20617452_20619962_0 . . . 234
8.1925 so_45813673_45817771_0 . . . 234
8.1926 so_28184984_28253270_0 . . . 234
8.1927 so_7254698_7257081_2 . . . 234
8.1928 so_29955404_29956343_1 . . . 234
8.1929 so_29571705_29571782_1 . . . 234
8.1930 so_16983539_16983926_1 . . . 234
8.1931 so_15032501_15032693_0 . . . 234
8.1932 so_19112399_19112572_2 . . . 234
8.1933 so_24678701_24679050_0 . . . 234
8.1934 so_17294925_17299103_1 . . . 234
8.1935 so_32521974_32522401_4 . . . 234
8.1936 so_43694112_43694276_2 . . . 234
8.1937 so_7215647_7372003_1 . . . 234
8.1938 so_31278377_31278474_6 . . . 234
8.1939 so_13886101_13886412_0 . . . 234
8.1940 so_13886101_13886302_1 . . . 234
8.1941 so_27669200_27669457_0 . . . 234
8.1942 so_39267963_39273340_5 . . . 234
8.1943 so_45551177_45551177_0 . . . 234
8.1944 so_27303650_27304057_1 . . . 234
8.1945 so_39577922_39577922_8 . . . 234

8.1946 so_44885143_44885326_1 . . . 234
8.1947 so_32940847_32946270_1 . . . 234
8.1948 so_32940847_32941074_0 . . . 234
8.1949 so_28058371_28058430_4 . . . 234
8.1950 so_21294831_28722054_0 . . . 234
8.1951 so_33828350_33829487_1 . . . 234
8.1952 so_31634962_31635036_2 . . . 234
8.1953 so_29970221_29970349_2 . . . 234
8.1954 so_26440801_26440829_1 . . . 234
8.1955 so_23892018_23965233_5 . . . 234
8.1956 so_23892018_23965233_4 . . . 234
8.1957 so_47315806_47316402_1 . . . 234
8.1958 so_21354522_21354634_0 . . . 234
8.1959 so_42309444_42309695_0 . . . 234
8.1960 so_23808357_23809231_1 . . . 234
8.1961 so_34856936_34857037_1 . . . 234
8.1962 so_32268576_32268576_0 . . . 234
8.1963 so_37298365_37298580_0 . . . 234
8.1964 so_23304025_23304090_3 . . . 234
8.1965 so_20641735_20963668_1 . . . 234
8.1966 so_18903113_18915868_14 . . . 234
8.1967 so_18903113_18915868_5 . . . 234
8.1968 so_36454463_36454463_2 . . . 234
8.1969 so_28074071_28074071_2 . . . 234
8.1970 so_14797810_14797812_0 . . . 234
8.1971 so_11394924_11394947_1 . . . 234
8.1972 so_29571705_29571782_2 . . . 235
8.1973 so_38082969_38083521_0 . . . 235
8.1974 so_38082969_38083521_1 . . . 235
8.1975 so_37391793_37391793_1 . . . 235
8.1976 so_24873891_24874210_6 . . . 235
8.1977 so_24873891_24874210_5 . . . 235
8.1978 so_40987645_40987705_0 . . . 235
8.1979 so_47435451_47435756_0 . . . 235
8.1980 so_2613562_2613578_0 . . . 235
8.1981 so_32912921_32912976_1 . . . 235
8.1982 so_22346426_22347985_1 . . . 235
8.1983 so_34856936_34857037_2 . . . 235
8.1984 so_35463646_35465666_1 . . . 235
8.1985 so_32576388_32576468_2 . . . 235
8.1986 so_27770814_27770873_3 . . . 235
8.1987 so_14797810_14797812_1 . . . 235
8.1988 so_9173322_9173805_4 . . . 235
8.1989 so_30249199_30249315_4 . . . 235
8.1990 so_7254698_7257081_3 . . . 235
8.1991 so_28408911_28409778_3 . . . 235
8.1992 so_20576162_20576364_0 . . . 235
8.1993 so_28112953_28113043_0 . . . 235
8.1994 so_35463646_35465666_3 . . . 235
8.1995 so_453432_453462_2 . . . 235
8.1996 so_24122006_24122006_4 . . . 235

8.1997 so_24122006_24122006_3 . . . 235
8.1998 so_8697105_8697182_2 . . . 235
8.1999 so_6284507_6284695_0 . . . 235
8.2000 so_25283151_25283388_0 . . . 235
8.2001 so_25949100_25949900_0 . . . 235
8.2002 so_29399870_29401520_2 . . . 235
8.2003 so_27382036_27388378_11 . . . 235
8.2004 so_29131416_29131794_0 . . . 235
8.2005 so_25964198_25964337_2 . . . 235
8.2006 so_28969309_28969576_0 . . . 235
8.2007 so_706030_706082_0 . . . 235
8.2008 so_706030_706082_1 . . . 235
8.2009 so_12164061_12164116_1 . . . 235
8.2010 so_27770814_27770873_5 . . . 235
8.2011 so_27770814_27770873_4 . . . 235
8.2012 so_5291008_5291024_2 . . . 235
8.2013 so_5291008_5291024_1 . . . 235
8.2014 so_39267963_39273340_8 . . . 235
8.2015 so_39267963_39273340_6 . . . 235
8.2016 so_12580171_12580210_1 . . . 235
8.2017 so_12580171_12580210_0 . . . 235
8.2018 so_24043577_24043750_3 . . . 235
8.2019 so_24043577_24043750_2 . . . 235
8.2020 so_24043577_24043804_0 . . . 235
8.2021 so_31271975_31272058_5 . . . 235
8.2022 so_31271975_31272058_6 . . . 235
8.2023 so_5610352_5610375_1 . . . 235
8.2024 so_5330672_5330710_2 . . . 235
8.2025 so_5330672_5330707_0 . . . 235
8.2026 so_5330672_5330707_1 . . . 235
8.2027 so_5330672_5330710_0 . . . 235
8.2028 so_3659839_3660859_0 . . . 235
8.2029 so_29291371_29292092_3 . . . 235
8.2030 so_4539563_4539588_0 . . . 235
8.2031 so_3580789_3580876_0 . . . 235
8.2032 so_39429510_39431986_3 . . . 235
8.2033 so_29487390_29495759_0 . . . 235
8.2034 so_14225545_14225556_0 . . . 235
8.2035 so_28113750_28113750_0 . . . 235
8.2036 so_37098922_37099162_0 . . . 235
8.2037 so_37098922_37099162_1 . . . 235
8.2038 so_26256174_26256294_2 . . . 235
8.2039 so_26256174_26256294_3 . . . 235
8.2040 so_6436554_6436615_0 . . . 235
8.2041 so_28675429_28675860_0 . . . 235
8.2042 so_22346426_22346962_1 . . . 235
8.2043 so_22346426_22346962_0 . . . 235
8.2044 so_42309444_42309695_1 . . . 235
8.2045 so_25804188_25804347_0 . . . 235
8.2046 so_26193718_26193854_2 . . . 235
8.2047 so_13097781_13097841_1 . . . 235

8.2048 so_13097781_13097841_0 . . . 235
8.2049 so_47096920_47097250_1 . . . 235
8.2050 so_10828294_10828333_4 . . . 235
8.2051 so_10828294_10828333_5 . . . 235
8.2052 so_88957_88960_2 . . . 235
8.2053 so_88957_88960_1 . . . 235
8.2054 so_41573959_41573959_0 . . . 235
8.2055 so_22849688_22849927_1 . . . 235
8.2056 so_13551572_13551661_1 . . . 235
8.2057 so_13551572_13551661_2 . . . 235
8.2058 so_18901315_18902462_3 . . . 235
8.2059 so_20936574_20936695_1 . . . 235
8.2060 so_20936574_20936695_0 . . . 235
8.2061 so_31091674_31091803_0 . . . 235
8.2062 so_14797810_14797812_2 . . . 235
8.2063 so_35449320_35449495_5 . . . 236
8.2064 so_28408911_28409778_10 . . . 236
8.2065 so_7254698_7257081_6 . . . 236
8.2066 so_22346426_22346899_2 . . . 236
8.2067 so_22346426_22346899_1 . . . 236
8.2068 so_22346426_22346899_3 . . . 236
8.2069 so_33852494_33852585_2 . . . 236
8.2070 so_33852494_33852585_1 . . . 236
8.2071 so_30249199_30249315_2 . . . 236
8.2072 so_30249199_30249315_3 . . . 236
8.2073 so_30249199_30249315_0 . . . 236
8.2074 so_26557231_26557488_2 . . . 236
8.2075 so_26557231_26557488_3 . . . 236
8.2076 so_20452967_20453967_1 . . . 236
8.2077 so_20452967_20453967_0 . . . 236
8.2078 so_20452967_20453967_2 . . . 236
8.2079 so_30959915_30959988_2 . . . 236
8.2080 so_30959915_30959988_0 . . . 236
8.2081 so_30959915_30959988_1 . . . 236
8.2082 so_11734861_11735045_3 . . . 236
8.2083 so_11734861_11735045_2 . . . 236
8.2084 so_11734861_11735045_4 . . . 236
8.2085 so_19092035_19092316_2 . . . 236
8.2086 so_19092035_19092316_3 . . . 236
8.2087 so_2687701_2687714_1 . . . 236
8.2088 so_41212740_41212992_0 . . . 236
8.2089 so_37421194_37421194_0 . . . 236
8.2090 so_37391793_37391793_0 . . . 236
8.2091 so_29593597_29593652_1 . . . 236
8.2092 so_11742592_11742888_5 . . . 236
8.2093 so_22346426_22346933_3 . . . 236
8.2094 so_22346426_22346899_7 . . . 236
8.2095 so_22346426_22346933_4 . . . 236
8.2096 so_22346426_22346899_6 . . . 236
8.2097 so_22346426_22346899_5 . . . 236
8.2098 so_22346426_22347985_0 . . . 236

8.2099 so_14684746_14696521_0 . . . 236
8.2100 so_40238484_40238583_6 . . . 236
8.2101 so_40238484_40238583_8 . . . 236
8.2102 so_29955404_29956343_0 . . . 237
8.2103 so_16985687_16985687_0 . . . 237
8.2104 so_2838107_2838143_0 . . . 237
8.2105 so_4283307_4283362_3 . . . 237
8.2106 so_39548254_39548345_1 . . . 237
8.2107 so_37618213_37618407_0 . . . 237
8.2108 so_39376813_39379359_1 . . . 237
8.2109 so_29357404_29357527_0 . . . 237
8.2110 so_31388208_31390380_8 . . . 237
8.2111 so_31638182_31638279_0 . . . 237
8.2112 so_16169161_16169240_8 . . . 237
8.2113 so_3031605_3031653_0 . . . 237
8.2114 so_35687736_35687838_0 . . . 237
8.2115 so_23493598_23493639_8 . . . 237
8.2116 so_46120191_46120385_0 . . . 238
8.2117 so_3524053_3524071_0 . . . 238
8.2118 so_4348154_4348188_0 . . . 238
8.2119 so_25898883_25899023_0 . . . 238
8.2120 so_31876378_31876401_0 . . . 238
8.2121 so_31876378_31876401_2 . . . 238
8.2122 so_29134443_29134572_2 . . . 238
8.2123 so_27770814_27770873_0 . . . 238
8.2124 so_26333381_26375498_0 . . . 238
8.2125 so_31550990_31552110_2 . . . 238
8.2126 so_31550990_31552110_1 . . . 238
8.2127 so_45551177_45551177_1 . . . 238
8.2128 so_21481925_21482059_0 . . . 238
8.2129 so_37861600_37862035_0 . . . 238
8.2130 so_26062865_26064070_1 . . . 238
8.2131 so_9069275_9069333_0 . . . 238
8.2132 so_10147264_38657078_1 . . . 238
8.2133 so_10147264_10147410_0 . . . 238
8.2134 so_17425372_17425429_1 . . . 238
8.2135 so_28433862_28433885_0 . . . 238
8.2136 so_13490805_13490908_0 . . . 238
8.2137 so_18218146_18218263_1 . . . 238
8.2138 so_8064616_8065714_1 . . . 238
8.2139 so_8064616_8064694_1 . . . 238
8.2140 so_8064616_8065714_2 . . . 238
8.2141 so_17790127_17790202_6 . . . 238
8.2142 so_17790127_17790202_8 . . . 238
8.2143 so_28114301_28114301_0 . . . 238
8.2144 so_3524053_3524071_2 . . . 238
8.2145 so_3524053_3524071_1 . . . 238
8.2146 so_27770814_27770873_1 . . . 238
8.2147 so_26333381_26375498_1 . . . 238
8.2148 so_37861600_37862035_1 . . . 238
8.2149 so_21951810_21951920_2 . . . 238

8.2150 so_28433862_28434100_1 . . . 238
8.2151 so_28433862_28433885_1 . . . 238
8.2152 so_13490805_13490908_7 . . . 238
8.2153 so_13490805_13490908_5 . . . 238
8.2154 so_27770814_27770873_2 . . . 238
8.2155 so_25362074_25362141_0 . . . 238
8.2156 so_17790127_17790202_10 . . . 238
8.2157 so_34388808_34389037_0 . . . 238
8.2158 so_28138376_28138436_6 . . . 238
8.2159 so_11794696_11794730_0 . . . 238
8.2160 so_25674834_25674834_3 . . . 238
8.2161 so_26113016_26113270_0 . . . 238
8.2162 so_15783380_15783431_0 . . . 238
8.2163 so_22781946_22782369_0 . . . 238
8.2164 so_18367698_18367907_1 . . . 238
8.2165 so_28138376_28138436_7 . . . 238
8.2166 so_8585783_8585783_0 . . . 238
8.2167 so_8804612_8808865_6 . . . 238
8.2168 so_15670136_15670920_0 . . . 238
8.2169 so_7389330_7389383_0 . . . 238
8.2170 so_22621380_22621380_0 . . . 238
8.2171 so_10781130_10781727_6 . . . 238
8.2172 so_28790233_28790233_1 . . . 238
8.2173 so_32942219_32942595_1 . . . 238
8.2174 so_31322845_31322959_0 . . . 239
8.2175 so_28138376_28138436_8 . . . 239
8.2176 so_18367698_18367907_3 . . . 239
8.2177 so_38953734_38953809_0 . . . 239
8.2178 so_10509603_10509736_5 . . . 239
8.2179 so_27675893_27676077_5 . . . 239
8.2180 so_11794696_11794730_3 . . . 239
8.2181 so_28371174_28371523_3 . . . 239
8.2182 so_20915139_20915233_0 . . . 239
8.2183 so_34640215_34640215_1 . . . 239
8.2184 so_15670136_15670920_1 . . . 239
8.2185 so_40006925_40010380_1 . . . 239
8.2186 so_12779630_12782315_0 . . . 239
8.2187 so_24657115_24657806_0 . . . 239
8.2188 so_27806504_27806823_0 . . . 239
8.2189 so_8598498_8598574_1 . . . 239
8.2190 so_15589842_15589842_4 . . . 239
8.2191 so_15589842_15589842_2 . . . 239
8.2192 so_21936668_21936668_0 . . . 239
8.2193 so_21936668_21936668_2 . . . 239
8.2194 so_21936668_21936668_1 . . . 239
8.2195 so_21936668_21936668_4 . . . 239
8.2196 so_21936668_21936668_3 . . . 239
8.2197 so_33619134_33621562_1 . . . 239
8.2198 so_33085796_33085881_0 . . . 239
8.2199 so_16723208_16723951_0 . . . 239
8.2200 so_16723208_16723951_2 . . . 239

8.2201 so_16723208_16723951_3 . . . 239
8.2202 so_13780417_34093903_0 . . . 239
8.2203 so_13780417_34093903_2 . . . 239
8.2204 so_34951161_34951347_0 . . . 239
8.2205 so_26901376_26901376_0 . . . 239
8.2206 so_26901376_26913217_1 . . . 239
8.2207 so_26901376_26901376_1 . . . 239
8.2208 so_26901376_26901376_3 . . . 239
8.2209 so_26901376_26901376_2 . . . 239
8.2210 so_34388808_34389037_1 . . . 239
8.2211 so_34388808_34389037_3 . . . 239
8.2212 so_33301552_33302605_11 . . . 239
8.2213 so_33872039_34017482_2 . . . 239
8.2214 so_33872039_34017482_4 . . . 239
8.2215 so_13898750_13898805_1 . . . 239
8.2216 so_32700860_32701150_3 . . . 239
8.2217 so_42043414_42043414_1 . . . 239
8.2218 so_19306984_19307380_0 . . . 239
8.2219 so_19306984_19307380_2 . . . 239
8.2220 so_23096929_23097164_0 . . . 239
8.2221 so_40540118_40540130_1 . . . 239
8.2222 so_27675893_27676077_1 . . . 239
8.2223 so_27675893_27676077_0 . . . 239
8.2224 so_34018046_34018046_1 . . . 239
8.2225 so_28371174_28371523_1 . . . 239
8.2226 so_28371174_28371523_2 . . . 239
8.2227 so_28371174_28371523_0 . . . 239
8.2228 so_26857674_26857732_1 . . . 239
8.2229 so_26857674_26857674_0 . . . 239
8.2230 so_26857674_26857732_0 . . . 239
8.2231 so_26857674_26857674_2 . . . 239
8.2232 so_26857674_26857732_3 . . . 239
8.2233 so_26857674_26857674_3 . . . 239
8.2234 so_26857674_26857674_5 . . . 239
8.2235 so_26857674_26857674_1 . . . 239
8.2236 so_22128890_22129113_3 . . . 239
8.2237 so_14935722_14991772_3 . . . 239
8.2238 so_21213881_21213881_0 . . . 239
8.2239 so_41505121_41505189_8 . . . 239
8.2240 so_41505121_41505189_9 . . . 239
8.2241 so_41505121_41505189_15 . . . 239
8.2242 so_41505121_41505189_11 . . . 239
8.2243 so_41505121_41505189_10 . . . 239
8.2244 so_41505121_41505189_13 . . . 239
8.2245 so_41505121_41505189_14 . . . 239
8.2246 so_41505121_41505189_12 . . . 239
8.2247 so_34898678_34898678_0 . . . 239
8.2248 so_27869719_27869778_2 . . . 239
8.2249 so_27869719_27869778_3 . . . 239
8.2250 so_27869719_27869778_4 . . . 239
8.2251 so_27869719_27869778_1 . . . 239

8.2252 so_27869719_27869778_0 . . . 239
8.2253 so_44698195_44701336_0 . . . 239
8.2254 so_30559817_30560064_0 . . . 239
8.2255 so_28176415_28176643_0 . . . 239
8.2256 so_28176415_28176643_1 . . . 239
8.2257 so_2748866_7230515_0 . . . 239
8.2258 so_17458698_17460139_0 . . . 239
8.2259 so_17458698_17460139_1 . . . 239
8.2260 so_16848222_16848222_0 . . . 239
8.2261 so_16848222_16848469_0 . . . 239
8.2262 so_33067600_33067600_27 . . . 239
8.2263 so_33067600_33067600_14 . . . 239
8.2264 so_33067600_33067600_35 . . . 239
8.2265 so_33067600_33068106_0 . . . 239
8.2266 so_33067600_33067600_24 . . . 239
8.2267 so_33067600_33067600_29 . . . 239
8.2268 so_33067600_33067600_23 . . . 239
8.2269 so_33067600_33067600_31 . . . 239
8.2270 so_33067600_33067600_8 . . . 239
8.2271 so_33067600_33067600_30 . . . 239
8.2272 so_33067600_33067600_10 . . . 239
8.2273 so_33067600_33067600_7 . . . 239
8.2274 so_33067600_33067600_15 . . . 239
8.2275 so_33067600_33067600_20 . . . 239
8.2276 so_33067600_33067600_6 . . . 239
8.2277 so_33067600_33067600_32 . . . 239
8.2278 so_33067600_33067600_21 . . . 239
8.2279 so_33067600_33067600_11 . . . 239
8.2280 so_33067600_33067600_4 . . . 239
8.2281 so_33067600_33067600_19 . . . 239
8.2282 so_33067600_33067600_0 . . . 239
8.2283 so_33067600_33067600_28 . . . 239
8.2284 so_35139114_35139114_1 . . . 239
8.2285 so_35139114_35140295_0 . . . 239
8.2286 so_3249147_3249527_7 . . . 239
8.2287 so_3249147_3249527_6 . . . 239
8.2288 so_3249147_3249527_1 . . . 239
8.2289 so_3249147_3249392_7 . . . 239
8.2290 so_37165073_37166407_1 . . . 239
8.2291 so_3848429_3848453_0 . . . 239
8.2292 so_24873891_24874210_1 . . . 239
8.2293 so_39797134_42889321_0 . . . 239
8.2294 so_6376580_6376764_1 . . . 239
8.2295 so_43129165_43129235_0 . . . 239
8.2296 so_13724553_13724838_2 . . . 239
8.2297 so_40193008_40193715_0 . . . 239
8.2298 so_40193008_40193715_2 . . . 239
8.2299 so_40193008_40193715_1 . . . 239
8.2300 so_40193008_40193546_0 . . . 239
8.2301 so_27678257_27678304_3 . . . 239
8.2302 so_17453738_28557688_2 . . . 239

8.2303 so_15670136_15670920_2 . . . 239
8.2304 so_15670136_15670920_3 . . . 239
8.2305 so_40006925_40010380_0 . . . 239
8.2306 so_40006925_40007311_1 . . . 239
8.2307 so_33074102_33074102_18 . . . 239
8.2308 so_33074102_33074102_16 . . . 239
8.2309 so_33074102_33074102_14 . . . 239
8.2310 so_33074102_33074102_12 . . . 239
8.2311 so_36349744_36350140_0 . . . 239
8.2312 so_27463785_27464124_1 . . . 239
8.2313 so_38933823_38936111_3 . . . 239
8.2314 so_5266783_5267706_2 . . . 239
8.2315 so_3232582_3233045_0 . . . 239
8.2316 so_3232582_3233045_2 . . . 239
8.2317 so_21916364_21916649_0 . . . 239
8.2318 so_21916364_21916364_3 . . . 239
8.2319 so_21916364_21916364_0 . . . 239
8.2320 so_21916364_21916364_1 . . . 239
8.2321 so_21916364_21916649_1 . . . 239
8.2322 so_21916364_21916364_2 . . . 239
8.2323 so_35947296_35947797_5 . . . 239
8.2324 so_35947296_35947797_2 . . . 239
8.2325 so_35947296_35947513_12 . . . 239
8.2326 so_35947296_35947513_6 . . . 239
8.2327 so_35947296_35947513_15 . . . 239
8.2328 so_35947296_35947572_1 . . . 239
8.2329 so_35947296_35947572_3 . . . 239
8.2330 so_35947296_35947797_10 . . . 239
8.2331 so_35947296_35947797_9 . . . 239
8.2332 so_35947296_35947797_7 . . . 239
8.2333 so_35947296_35947797_1 . . . 239
8.2334 so_35947296_35947797_0 . . . 239
8.2335 so_35947296_35947513_7 . . . 239
8.2336 so_35947296_35947513_5 . . . 239
8.2337 so_35947296_35947513_10 . . . 239
8.2338 so_35947296_35947513_14 . . . 239
8.2339 so_32784399_32852344_1 . . . 239
8.2340 so_2315398_39835100_0 . . . 239
8.2341 so_15938077_15956145_4 . . . 239
8.2342 so_22885613_22885737_2 . . . 239
8.2343 so_22885613_22885737_0 . . . 239
8.2344 so_22885613_22885737_1 . . . 239
8.2345 so_12779630_12782315_2 . . . 239
8.2346 so_12779630_12782315_4 . . . 239
8.2347 so_12779630_12782315_1 . . . 239
8.2348 so_14298715_14300750_8 . . . 239
8.2349 so_14298715_14300750_6 . . . 239
8.2350 so_14298715_14300750_3 . . . 239
8.2351 so_14298715_14300750_2 . . . 239
8.2352 so_14298715_14300750_0 . . . 239
8.2353 so_34901154_34901154_2 . . . 239

8.2354 so_34901154_34901154_3 . . . 239
8.2355 so_25395297_25395408_10 . . . 239
8.2356 so_25395297_25395408_6 . . . 239
8.2357 so_25395297_25395408_12 . . . 239
8.2358 so_25395297_25395408_8 . . . 239
8.2359 so_25395297_25395408_11 . . . 239
8.2360 so_25395297_25395408_7 . . . 239
8.2361 so_25395297_25395408_9 . . . 239
8.2362 so_28048208_28048551_0 . . . 239
8.2363 so_27840925_27881306_7 . . . 239
8.2364 so_27840925_27840925_0 . . . 239
8.2365 so_27840925_27881306_8 . . . 239
8.2366 so_27840925_27881306_6 . . . 239
8.2367 so_27840925_27840925_1 . . . 239
8.2368 so_27840925_27881306_9 . . . 239
8.2369 so_37718495_37718726_3 . . . 239
8.2370 so_37718495_37718726_1 . . . 239
8.2371 so_31187325_31187325_0 . . . 239
8.2372 so_34160614_34162106_0 . . . 239
8.2373 so_34160614_34162106_2 . . . 239
8.2374 so_34160614_34162106_1 . . . 239
8.2375 so_4808922_4809101_0 . . . 239
8.2376 so_17428394_17437848_1 . . . 239
8.2377 so_1615660_1615812_0 . . . 239
8.2378 so_1615660_1624719_0 . . . 239
8.2379 so_1615660_1624719_3 . . . 239
8.2380 so_31526865_31527020_4 . . . 239
8.2381 so_31526865_31527020_5 . . . 239
8.2382 so_26724674_26725089_3 . . . 239
8.2383 so_21045518_21046043_3 . . . 239
8.2384 so_21045518_21046043_2 . . . 239
8.2385 so_4097762_4097876_3 . . . 239
8.2386 so_17127652_17127680_0 . . . 239
8.2387 so_16493524_16495384_11 . . . 239
8.2388 so_16493524_16495384_9 . . . 239
8.2389 so_16493524_16495384_5 . . . 239
8.2390 so_16493524_16495384_4 . . . 239
8.2391 so_16493524_16495384_7 . . . 239
8.2392 so_16493524_16495384_3 . . . 239
8.2393 so_16493524_16495384_12 . . . 239
8.2394 so_16493524_16495384_6 . . . 239
8.2395 so_24072180_24072309_2 . . . 239
8.2396 so_25745759_25758367_4 . . . 239
8.2397 so_25745759_25758367_6 . . . 239
8.2398 so_25745759_25758367_5 . . . 239
8.2399 so_12881209_12881963_0 . . . 239
8.2400 so_27482261_27483146_4 . . . 239
8.2401 so_27482261_27483146_6 . . . 239
8.2402 so_27482261_27483146_10 . . . 239
8.2403 so_27482261_27483146_9 . . . 239
8.2404 so_27482261_27483146_0 . . . 239

8.2405 so_27482261_27483146_5 . . . 239
8.2406 so_27482261_27483146_3 . . . 239
8.2407 so_27482261_27483146_8 . . . 239
8.2408 so_27482261_27483146_2 . . . 239
8.2409 so_27482261_27483146_7 . . . 239
8.2410 so_27482261_27483146_1 . . . 239
8.2411 so_23610117_23610649_0 . . . 239
8.2412 so_29345522_29345946_0 . . . 239
8.2413 so_29345522_29345946_2 . . . 239
8.2414 so_27950769_27950769_2 . . . 239
8.2415 so_24943093_24943784_0 . . . 239
8.2416 so_5689463_5689686_2 . . . 239
8.2417 so_5689463_5689686_0 . . . 239
8.2418 so_26954276_26954276_1 . . . 239
8.2419 so_26954276_26954276_4 . . . 239
8.2420 so_26954276_26955983_8 . . . 239
8.2421 so_26954276_26954276_2 . . . 239
8.2422 so_26954276_26954276_5 . . . 239
8.2423 so_26954276_26954276_3 . . . 239
8.2424 so_17608637_17608907_0 . . . 239
8.2425 so_8245027_8246688_0 . . . 239
8.2426 so_8245027_8246688_3 . . . 239
8.2427 so_27806504_27806823_1 . . . 239
8.2428 so_27806504_27806823_3 . . . 239
8.2429 so_27806504_27806823_2 . . . 239
8.2430 so_42742995_42743013_0 . . . 239
8.2431 so_39277869_39278520_0 . . . 239
8.2432 so_18903113_18915868_6 . . . 239
8.2433 so_23685460_23685580_0 . . . 239
8.2434 so_33855956_33855956_0 . . . 239
8.2435 so_33855956_33855956_1 . . . 239
8.2436 so_31896298_31897188_1 . . . 241
8.2437 so_13461027_13461027_0 . . . 241
8.2438 so_24720028_24720311_3 . . . 241
8.2439 so_17381573_17381687_0 . . . 241
8.2440 so_11420448_11445905_0 . . . 241
8.2441 so_32521974_32522401_1 . . . 241
8.2442 so_13693871_13693871_7 . . . 241
8.2443 so_38131945_38131963_2 . . . 241
8.2444 so_2357452_2357688_5 . . . 241
8.2445 so_33913572_33913851_0 . . . 241
8.2446 so_12911527_12911541_1 . . . 241
8.2447 so_18215277_18215991_1 . . . 241
8.2448 so_39267963_39273340_3 . . . 241
8.2449 so_32648805_32652158_1 . . . 241
8.2450 so_35273025_35273245_0 . . . 241
8.2451 so_22458393_22459120_0 . . . 241
8.2452 so_17490843_17490934_0 . . . 241
8.2453 so_14871081_14871112_0 . . . 241
8.2454 so_23493598_23493639_2 . . . 241
8.2455 so_36454463_36454463_0 . . . 241

8.2456 so_28838286_28838379_1 . . . 241
8.2457 so_15286450_15286450_1 . . . 241
8.2458 so_34912221_34912221_3 . . . 242
8.2459 so_31211021_31216487_0 . . . 242
8.2460 so_1863784_1870069_1 . . . 242
8.2461 so_2357452_2357688_6 . . . 242
8.2462 so_8762459_8762706_0 . . . 242
8.2463 so_26414443_26414748_0 . . . 242
8.2464 so_30971502_30971502_0 . . . 242
8.2465 so_15641318_15641365_0 . . . 242
8.2466 so_46070793_46071137_1 . . . 242
8.2467 so_26426222_26426367_0 . . . 242
8.2468 so_32270244_32270347_1 . . . 242
8.2469 so_45880152_45880569_0 . . . 242
8.2470 so_38935256_38966560_0 . . . 242
8.2471 so_9173322_9173805_5 . . . 242
8.2472 so_19737929_19738076_3 . . . 242
8.2473 so_19737929_19738076_4 . . . 242
8.2474 so_31896298_31897188_7 . . . 242
8.2475 so_46339519_46339888_0 . . . 242
8.2476 so_39005700_39005787_1 . . . 242
8.2477 so_39005700_39005700_2 . . . 242
8.2478 so_39005700_39005700_1 . . . 242
8.2479 so_21210626_21210693_0 . . . 242
8.2480 so_10509603_10509736_6 . . . 242
8.2481 so_10509603_10509736_7 . . . 242
8.2482 so_10509603_10509736_3 . . . 242
8.2483 so_10509603_10509736_1 . . . 242
8.2484 so_10509603_10509736_2 . . . 242
8.2485 so_7254698_7257081_4 . . . 242
8.2486 so_28408911_28409778_1 . . . 242
8.2487 so_29571705_29571768_0 . . . 242
8.2488 so_29571705_29571768_1 . . . 242
8.2489 so_13459057_13460468_1 . . . 242
8.2490 so_26964221_26964314_0 . . . 242
8.2491 so_31685132_31685448_2 . . . 242
8.2492 so_8578791_8579314_1 . . . 242
8.2493 so_22128890_22129113_0 . . . 242
8.2494 so_22128890_22129113_1 . . . 242
8.2495 so_22128890_22129113_2 . . . 242
8.2496 so_27090093_27090513_6 . . . 242
8.2497 so_27090093_27090513_5 . . . 242
8.2498 so_27090093_27090513_3 . . . 242
8.2499 so_24122006_24122006_2 . . . 242
8.2500 so_3541661_3541674_1 . . . 242
8.2501 so_32521974_32522401_3 . . . 242
8.2502 so_32521974_32522401_2 . . . 242
8.2503 so_38131945_38131963_3 . . . 242
8.2504 so_25949100_25949900_1 . . . 242
8.2505 so_42586836_42587200_1 . . . 242
8.2506 so_42586836_42587200_3 . . . 242

8.2507 so_16985687_16985687_1 . . . 242
8.2508 so_27382036_27388378_9 . . . 242
8.2509 so_27382036_27388378_12 . . . 242
8.2510 so_27382036_27388378_13 . . . 242
8.2511 so_27382036_27388378_10 . . . 242
8.2512 so_27645671_27645680_1 . . . 242
8.2513 so_27645671_27645680_2 . . . 242
8.2514 so_33988297_33988299_1 . . . 242
8.2515 so_26788402_26788402_0 . . . 242
8.2516 so_40807404_40807404_1 . . . 242
8.2517 so_2357452_2357688_10 . . . 242
8.2518 so_37855300_37857288_0 . . . 242
8.2519 so_33970892_33978450_3 . . . 242
8.2520 so_33970892_33970892_0 . . . 242
8.2521 so_29131416_29131794_1 . . . 242
8.2522 so_25964198_25964400_1 . . . 242
8.2523 so_25964198_25964337_1 . . . 242
8.2524 so_40018076_40019058_0 . . . 242
8.2525 so_31278377_31278474_1 . . . 242
8.2526 so_31278377_31278474_3 . . . 242
8.2527 so_31278377_31278474_0 . . . 242
8.2528 so_28969309_28969576_1 . . . 242
8.2529 so_28717682_28718438_0 . . . 242
8.2530 so_28717682_28718438_2 . . . 242
8.2531 so_31776359_31776975_2 . . . 242
8.2532 so_31776359_31776975_0 . . . 242
8.2533 so_27844971_27877127_8 . . . 242
8.2534 so_27844971_27877127_6 . . . 242
8.2535 so_6546470_6547589_0 . . . 242
8.2536 so_36312411_36313320_3 . . . 242
8.2537 so_24873891_24874210_2 . . . 242
8.2538 so_24873891_24874210_3 . . . 242
8.2539 so_33913572_33913851_1 . . . 242
8.2540 so_16939471_16939798_0 . . . 242
8.2541 so_16939471_16939798_1 . . . 242
8.2542 so_16939471_16939798_4 . . . 242
8.2543 so_42663166_42675964_0 . . . 242
8.2544 so_13423007_13423263_1 . . . 242
8.2545 so_28391796_28391896_0 . . . 242
8.2546 so_28006980_28007010_0 . . . 242
8.2547 so_24794598_24794857_1 . . . 242
8.2548 so_24794598_24794857_0 . . . 242
8.2549 so_19574351_19574935_1 . . . 242
8.2550 so_19574351_19574935_0 . . . 242
8.2551 so_19574351_19574913_0 . . . 242
8.2552 so_19574351_19574935_2 . . . 242
8.2553 so_28128174_28128270_0 . . . 242
8.2554 so_28128174_28128270_1 . . . 242
8.2555 so_32032249_32034107_0 . . . 242
8.2556 so_31036066_31036612_0 . . . 242
8.2557 so_18895737_18896048_0 . . . 242

8.2558 so_39432311_39434234_2 . . . 242
8.2559 so_39432311_39434234_0 . . . 242
8.2560 so_21481925_21482157_1 . . . 242
8.2561 so_27303650_27304057_2 . . . 242
8.2562 so_39577922_39577922_6 . . . 242
8.2563 so_39577922_39577922_0 . . . 242
8.2564 so_39577922_39577922_5 . . . 242
8.2565 so_39577922_39577922_1 . . . 242
8.2566 so_39577922_39577922_7 . . . 242
8.2567 so_32648805_32652158_2 . . . 242
8.2568 so_23047198_23063914_2 . . . 242
8.2569 so_23047198_23063914_1 . . . 242
8.2570 so_41248340_41249238_4 . . . 242
8.2571 so_41248340_41249238_2 . . . 242
8.2572 so_26699720_26700151_4 . . . 242
8.2573 so_26699720_26700151_1 . . . 242
8.2574 so_38757079_38757329_0 . . . 242
8.2575 so_38757079_38757329_2 . . . 242
8.2576 so_33479555_33479868_4 . . . 242
8.2577 so_33479555_33479868_6 . . . 242
8.2578 so_32940847_32941074_3 . . . 242
8.2579 so_35273025_35273245_5 . . . 242
8.2580 so_35273025_35273245_1 . . . 242
8.2581 so_35273025_35273245_6 . . . 242
8.2582 so_35273025_35273245_2 . . . 242
8.2583 so_19331508_19331742_3 . . . 242
8.2584 so_19331508_19331742_2 . . . 242
8.2585 so_21825933_21836197_15 . . . 242
8.2586 so_21825933_21836197_16 . . . 242
8.2587 so_21825933_21836197_14 . . . 242
8.2588 so_21825933_21836197_18 . . . 242
8.2589 so_21825933_21836197_20 . . . 242
8.2590 so_21825933_21836197_21 . . . 242
8.2591 so_21825933_21836197_19 . . . 242
8.2592 so_21825933_21836197_17 . . . 242
8.2593 so_29291371_29292092_1 . . . 242
8.2594 so_17382796_17382796_0 . . . 242
8.2595 so_36152020_36163092_0 . . . 242
8.2596 so_39588505_39588505_0 . . . 242
8.2597 so_39588505_39590178_0 . . . 242
8.2598 so_39588505_39590178_1 . . . 242
8.2599 so_39588505_39588505_4 . . . 242
8.2600 so_29440617_29441011_0 . . . 242
8.2601 so_31728686_31728949_0 . . . 242
8.2602 so_23882409_23883035_15 . . . 242
8.2603 so_23882409_23883035_9 . . . 242
8.2604 so_23882409_23883035_0 . . . 242
8.2605 so_23882409_23883036_0 . . . 242
8.2606 so_23882409_23883035_1 . . . 242
8.2607 so_23882409_23883035_12 . . . 242
8.2608 so_23882409_23883035_13 . . . 242

8.2609 so_23882409_23883035_10 . . . 242
8.2610 so_39429510_39431986_0 . . . 242
8.2611 so_39429510_39431986_1 . . . 242
8.2612 so_27507361_27507361_0 . . . 242
8.2613 so_27507361_27508570_4 . . . 242
8.2614 so_29970221_29970349_0 . . . 242
8.2615 so_26440801_26440829_0 . . . 242
8.2616 so_42534992_42535090_0 . . . 242
8.2617 so_26256174_26256294_1 . . . 242
8.2618 so_26256174_26256294_4 . . . 242
8.2619 so_26256174_26256294_0 . . . 242
8.2620 so_26256174_26256294_5 . . . 242
8.2621 so_26567969_26571935_2 . . . 242
8.2622 so_26567969_26571935_0 . . . 242
8.2623 so_25804188_25804347_1 . . . 242
8.2624 so_24718555_24719013_0 . . . 242
8.2625 so_44306607_44310738_4 . . . 242
8.2626 so_29683381_29683772_2 . . . 242
8.2627 so_32270244_32270723_0 . . . 242
8.2628 so_32270244_32270347_4 . . . 242
8.2629 so_21340801_21341177_1 . . . 242
8.2630 so_12596158_12596333_2 . . . 242
8.2631 so_15126495_15126673_4 . . . 242
8.2632 so_15126495_15126673_6 . . . 242
8.2633 so_15126495_15126673_0 . . . 242
8.2634 so_15126495_15126673_1 . . . 242
8.2635 so_15126495_15126673_3 . . . 242
8.2636 so_15126495_15126673_5 . . . 242
8.2637 so_19121890_19122919_12 . . . 242
8.2638 so_19121890_19122919_15 . . . 242
8.2639 so_19121890_19122919_13 . . . 242
8.2640 so_23304025_23304090_0 . . . 242
8.2641 so_26640866_26641063_0 . . . 242
8.2642 so_26640866_26641063_1 . . . 242
8.2643 so_28466069_28466464_1 . . . 242
8.2644 so_22849688_22849927_0 . . . 242
8.2645 so_38061779_38065158_0 . . . 242
8.2646 so_27237423_27237787_1 . . . 242
8.2647 so_21708606_21708950_0 . . . 242
8.2648 so_21708606_21708950_1 . . . 242
8.2649 so_21708606_21710850_5 . . . 242
8.2650 so_30461583_30461995_0 . . . 242
8.2651 so_18901315_18902462_2 . . . 242
8.2652 so_20936574_20936695_2 . . . 242
8.2653 so_33297916_33298199_0 . . . 242
8.2654 so_11449210_11449654_1 . . . 242
8.2655 so_28074071_28074354_1 . . . 242
8.2656 so_33832423_33832423_0 . . . 242
8.2657 so_19347004_19347160_0 . . . 242
8.2658 so_19347004_19347160_1 . . . 242
8.2659 so_19347004_19347160_3 . . . 242

8.2660 so_11394924_11396061_1 . . . 242
8.2661 so_15286450_15286450_3 . . . 242
8.2662 so_15286450_15286450_4 . . . 242
8.2663 so_26947704_26949366_0 . . . 242
8.2664 so_26947704_26949366_1 . . . 242
8.2665 so_16017767_16018026_0 . . . 244
8.2666 so_45610376_45610924_0 . . . 244
8.2667 so_24720028_24720119_2 . . . 244
8.2668 so_17381573_17381687_1 . . . 244
8.2669 so_17381573_17381573_0 . . . 244
8.2670 so_33620783_33621132_0 . . . 244
8.2671 so_12048221_12048366_1 . . . 244
8.2672 so_37667952_37668941_0 . . . 244
8.2673 so_37254813_37254813_0 . . . 244
8.2674 so_36852367_36852367_1 . . . 244
8.2675 so_24326572_24327094_3 . . . 244
8.2676 so_24326572_24326720_0 . . . 244
8.2677 so_41847440_41847440_0 . . . 244
8.2678 so_24224115_24226345_6 . . . 244
8.2679 so_4114794_4160654_0 . . . 244
8.2680 so_14060264_14060264_1 . . . 244
8.2681 so_28106056_28106465_0 . . . 244
8.2682 so_24289069_24289162_0 . . . 244
8.2683 so_24814696_24814759_0 . . . 244
8.2684 so_21234380_21234522_0 . . . 244
8.2685 so_31333729_31333886_0 . . . 244
8.2686 so_32940847_32945791_1 . . . 244
8.2687 so_32940847_32945791_3 . . . 244
8.2688 so_32940847_32941074_1 . . . 244
8.2689 so_22515685_22515736_2 . . . 244
8.2690 so_22515685_22515948_0 . . . 244
8.2691 so_22515685_22515830_0 . . . 244
8.2692 so_38501587_38501596_10 . . . 244
8.2693 so_21294831_28722054_1 . . . 244
8.2694 so_19861927_19862240_2 . . . 244
8.2695 so_26918903_26919328_4 . . . 244
8.2696 so_39340030_39340191_1 . . . 244
8.2697 so_39340030_39340191_0 . . . 244
8.2698 so_20266153_20266642_0 . . . 244
8.2699 so_9467868_9468380_3 . . . 244
8.2700 so_20186084_20186870_0 . . . 244
8.2701 so_18731707_29337570_3 . . . 244
8.2702 so_15755632_15756233_0 . . . 244
8.2703 so_31724863_31725109_0 . . . 244
8.2704 so_31724863_31725109_2 . . . 244
8.2705 so_31724863_31725109_1 . . . 244
8.2706 so_24793019_24793143_0 . . . 244
8.2707 so_44762154_44762820_0 . . . 244
8.2708 so_29200036_29200036_0 . . . 244
8.2709 so_18686991_18687162_0 . . . 244
8.2710 so_36377758_36377968_0 . . . 244

8.2711 so_37738868_37738978_0 . . . 244
8.2712 so_8762459_8762706_1 . . . 244
8.2713 so_26021209_28245830_0 . . . 244
8.2714 so_39432311_39434234_1 . . . 244
8.2715 so_21884499_21884973_0 . . . 244
8.2716 so_21884499_21884973_1 . . . 244
8.2717 so_19066974_19067212_0 . . . 244
8.2718 so_25330525_25330821_1 . . . 244
8.2719 so_21223992_21224099_1 . . . 244
8.2720 so_7231351_7232135_0 . . . 244
8.2721 so_11174293_11174389_0 . . . 244
8.2722 so_24752830_24753189_0 . . . 244
8.2723 so_24752830_24753189_2 . . . 244
8.2724 so_24752830_24753189_1 . . . 244
8.2725 so_28774722_28775143_0 . . . 244
8.2726 so_25827435_25827463_0 . . . 244
8.2727 so_27827720_33489694_0 . . . 244
8.2728 so_45547655_45548083_0 . . . 244
8.2729 so_45880152_45880569_1 . . . 244
8.2730 so_15286450_15286450_5 . . . 244
8.2731 so_44762154_44762820_1 . . . 245
8.2732 so_44762154_44762820_2 . . . 245
8.2733 so_5840810_5840907_0 . . . 245
8.2734 so_16349006_16351481_0 . . . 245
8.2735 so_21884499_21884973_3 . . . 245
8.2736 so_19066974_19067212_1 . . . 245
8.2737 so_28318799_28323698_10 . . . 245
8.2738 so_28318799_28323698_1 . . . 245
8.2739 so_15937522_15938175_0 . . . 245
8.2740 so_11174293_11174389_1 . . . 245
8.2741 so_28774722_28775143_1 . . . 245
8.2742 so_28774722_28775143_2 . . . 245
8.2743 so_30735730_30737281_0 . . . 245
8.2744 so_45547655_45548083_1 . . . 245
8.2745 so_15286450_15286450_7 . . . 245
8.2746 so_15286450_17635374_1 . . . 245
8.2747 so_15286450_15286450_6 . . . 245
8.2748 so_15286450_17635374_0 . . . 245
8.2749 so_19737929_19737945_0 . . . 245
8.2750 so_19737929_19738076_1 . . . 245
8.2751 so_38960126_38961143_2 . . . 245
8.2752 so_40805865_40805946_0 . . . 245
8.2753 so_40805865_40805946_1 . . . 245
8.2754 so_8251059_8251081_0 . . . 245
8.2755 so_31685132_31685448_6 . . . 245
8.2756 so_11269623_11269907_1 . . . 245
8.2757 so_21150067_21169201_4 . . . 245
8.2758 so_21150067_21169201_5 . . . 245
8.2759 so_21150067_21169201_3 . . . 245
8.2760 so_21150067_21169201_2 . . . 245
8.2761 so_15297191_15297282_0 . . . 245

8.2762 so_40807404_40807404_0 . . . 245
8.2763 so_17551583_17551583_5 . . . 245
8.2764 so_17551583_17551583_1 . . . 245
8.2765 so_17551583_17551583_4 . . . 245
8.2766 so_17551583_17551583_3 . . . 245
8.2767 so_17551583_17551583_2 . . . 245
8.2768 so_27844971_27877127_5 . . . 245
8.2769 so_19643143_19643369_2 . . . 245
8.2770 so_11521016_11521166_1 . . . 245
8.2771 so_41253121_41253658_0 . . . 245
8.2772 so_41248340_41248520_0 . . . 245
8.2773 so_41248340_41248530_0 . . . 245
8.2774 so_24345399_24346350_2 . . . 245
8.2775 so_21242070_21242231_0 . . . 245
8.2776 so_28094263_28094380_0 . . . 245
8.2777 so_31634962_31635036_1 . . . 245
8.2778 so_33926029_33926029_1 . . . 245
8.2779 so_44422223_44423522_0 . . . 245
8.2780 so_27507361_27508570_3 . . . 245
8.2781 so_27507361_27507361_1 . . . 245
8.2782 so_26974775_26974911_1 . . . 245
8.2783 so_33605632_33606398_3 . . . 245
8.2784 so_17359235_17359774_0 . . . 245
8.2785 so_45875000_45875324_0 . . . 245
8.2786 so_39463323_39463377_2 . . . 245
8.2787 so_39463323_39463377_5 . . . 245
8.2788 so_39463323_39463377_6 . . . 245
8.2789 so_23441180_23442128_0 . . . 245

9 Classes **247**
9.1 so_40917441_40917441_1 . . . 247
9.2 so_14248044_14248044_2 . . . 247
9.3 so_7371020_7371035_3 . . . 247
9.4 so_7371020_7371035_4 . . . 247
9.5 so_5337718_5337718_1 . . . 247
9.6 so_5337718_5337718_0 . . . 247
9.7 so_5337718_5337718_3 . . . 247
9.8 so_29133988_29136447_0 . . . 247
9.9 so_12843893_12844039_0 . . . 247
9.10 so_47546616_47546665_1 . . . 247
9.11 so_16127770_16127941_0 . . . 247
9.12 so_32037146_32037250_0 . . . 247
9.13 so_32037146_32037243_0 . . . 247
9.14 so_37652083_37652168_0 . . . 247
9.15 so_7025054_7025054_1 . . . 247
9.16 so_12628725_12628805_0 . . . 247
9.17 so_25038479_25038543_0 . . . 247
9.18 so_14227721_14227763_0 . . . 247
9.19 so_33350002_33350762_2 . . . 247
9.20 so_31177609_32401503_0 . . . 247
9.21 so_10050863_10050964_0 . . . 247

9.22 so_11423380_11423718_0 . . . 247
9.23 so_4277477_4277523_0 . . . 247
9.24 so_27152477_27152678_0 . . . 247
9.25 so_29705794_29705836_0 . . . 247
9.26 so_35162397_35162505_0 . . . 247
9.27 so_30155802_30156450_3 . . . 247
9.28 so_30155802_30156450_0 . . . 247
9.29 so_17673778_17673947_1 . . . 247
9.30 so_3546340_3546772_0 . . . 247
9.31 so_36545819_36548889_0 . . . 247
9.32 so_32602805_32603265_0 . . . 247
9.33 so_25549652_25549691_1 . . . 247
9.34 so_25549652_25549691_0 . . . 247
9.35 so_15342410_15342557_0 . . . 247
9.36 so_44548242_44549084_2 . . . 247
9.37 so_31769853_31769853_0 . . . 247
9.38 so_19886094_19886144_0 . . . 247
9.39 so_32006122_32006122_0 . . . 247
9.40 so_47575074_47575256_0 . . . 247
9.41 so_12135498_12135512_0 . . . 247
9.42 so_36895012_36898976_1 . . . 247
9.43 so_29710183_29710305_0 . . . 247
9.44 so_21915630_21917031_0 . . . 248
9.45 so_4623963_4623963_0 . . . 248
9.46 so_8271673_8271750_0 . . . 248
9.47 so_18551174_18551259_0 . . . 248
9.48 so_24865741_24865832_1 . . . 248
9.49 so_39763548_39764671_0 . . . 248
9.50 so_23716391_23716762_3 . . . 248
9.51 so_18806954_18807039_2 . . . 248
9.52 so_14543983_14544093_1 . . . 248
9.53 so_32997185_32997605_4 . . . 248
9.54 so_4178175_7189821_9 . . . 248
9.55 so_4178175_7189821_8 . . . 248
9.56 so_4178175_7189821_6 . . . 248
9.57 so_4178175_7189821_5 . . . 248
9.58 so_30096911_30097049_1 . . . 248
9.59 so_25830995_25831292_1 . . . 248
9.60 so_5430801_5453372_2 . . . 248
9.61 so_5430801_5453372_1 . . . 248
9.62 so_12979529_12979623_3 . . . 248
9.63 so_36098055_36098055_5 . . . 248
9.64 so_36098055_36098055_4 . . . 248
9.65 so_11979153_11979192_3 . . . 248
9.66 so_11979153_11979192_4 . . . 248
9.67 so_4275861_4417899_5 . . . 248
9.68 so_3674247_3679872_1 . . . 248
9.69 so_7536153_7536185_1 . . . 248
9.70 so_18901315_18902462_1 . . . 248
9.71 so_16542291_16542385_6 . . . 248
9.72 so_16542291_16542385_4 . . . 248

9.73 so_16542291_16542385_7 . . . 248
9.74 so_16542291_16542385_3 . . . 248
9.75 so_29915596_29916211_1 . . . 248
9.76 so_31398388_31398779_1 . . . 248
9.77 so_26754024_26754247_1 . . . 248
9.78 so_13900421_13900462_0 . . . 248
9.79 so_26939609_26939661_5 . . . 248
9.80 so_3820363_3820388_8 . . . 248
9.81 so_3820363_3820388_7 . . . 248
9.82 so_3820363_3820388_5 . . . 248
9.83 so_3820363_3820388_3 . . . 248
9.84 so_4478983_4479183_1 . . . 248
9.85 so_4178175_7189821_16 . . . 248
9.86 so_4178175_9491406_1 . . . 248
9.87 so_4178175_7189821_20 . . . 248
9.88 so_4178175_7189821_18 . . . 248
9.89 so_4178175_7189821_21 . . . 248
9.90 so_13861161_13861611_6 . . . 248
9.91 so_13861161_13861611_7 . . . 248
9.92 so_13861161_13861611_4 . . . 248
9.93 so_13861161_13861611_2 . . . 248
9.94 so_13861161_13862236_1 . . . 248
9.95 so_10788823_10788823_0 . . . 248
9.96 so_10788823_10788823_2 . . . 248
9.97 so_10788823_10788823_5 . . . 248
9.98 so_10788823_10788823_4 . . . 248
9.99 so_12979529_12979623_2 . . . 248
9.100 so_29930982_29931010_2 . . . 248
9.101 so_11340028_11340165_0 . . . 248
9.102 so_36354041_36354149_3 . . . 248
9.103 so_36354041_36354145_1 . . . 248
9.104 so_36354041_36354041_5 . . . 248
9.105 so_36354041_36354041_2 . . . 248
9.106 so_36354041_36354149_6 . . . 248
9.107 so_36354041_36354149_1 . . . 248
9.108 so_36354041_36354149_5 . . . 248
9.109 so_36354041_36354041_3 . . . 248
9.110 so_14941723_14941807_4 . . . 248
9.111 so_3902513_3902513_0 . . . 248
9.112 so_17845128_17845164_0 . . . 248
9.113 so_29280690_29280690_1 . . . 248
9.114 so_37323232_37369101_4 . . . 248
9.115 so_22404423_22404659_0 . . . 248
9.116 so_19843816_19868173_8 . . . 248
9.117 so_19843816_19868173_9 . . . 248
9.118 so_19843816_19868173_4 . . . 248
9.119 so_19843816_19868173_6 . . . 248
9.120 so_2624628_2624947_7 . . . 248
9.121 so_2624628_2624947_2 . . . 248
9.122 so_2624628_2624947_6 . . . 248
9.123 so_2624628_2624947_4 . . . 248

9.124 so_26529842_26529842_1 . . . 248
9.125 so_4178175_7189821_23 . . . 248
9.126 so_4178175_7189821_22 . . . 248
9.127 so_13541325_13543222_2 . . . 248
9.128 so_20743582_20743681_2 . . . 248
9.129 so_7762929_7762964_2 . . . 248
9.130 so_7762929_7762964_3 . . . 248
9.131 so_32719880_41274670_0 . . . 248
9.132 so_32719880_35698862_1 . . . 248
9.133 so_29280690_29280690_2 . . . 248
9.134 so_43988835_43989740_2 . . . 248
9.135 so_37323232_37369101_2 . . . 248
9.136 so_19843816_19868173_10 . . . 248
9.137 so_2624628_2624947_8 . . . 248
9.138 so_34906229_34906229_1 . . . 248
9.139 so_31398388_31398779_3 . . . 248
9.140 so_4178175_7189821_24 . . . 248
9.141 so_36354041_36354145_0 . . . 248
9.142 so_32719880_41274670_4 . . . 248
9.143 so_32719880_35698862_2 . . . 248
9.144 so_19843816_19868173_11 . . . 248
9.145 so_23716391_23716762_4 . . . 248
9.146 so_23716391_23716762_5 . . . 248
9.147 so_18806954_18807039_3 . . . 248
9.148 so_14543983_14544093_0 . . . 248
9.149 so_4178175_27511360_2 . . . 248
9.150 so_4178175_7189821_1 . . . 248
9.151 so_12979529_12979623_1 . . . 248
9.152 so_42078863_42079254_0 . . . 248
9.153 so_3952020_3952020_1 . . . 248
9.154 so_29930982_29931010_4 . . . 248
9.155 so_21074892_21074979_0 . . . 248
9.156 so_3674247_3674414_1 . . . 248
9.157 so_32719880_41274670_1 . . . 248
9.158 so_7624714_7624757_0 . . . 248
9.159 so_22404423_22404543_0 . . . 248
9.160 so_19590008_19590429_0 . . . 248
9.161 so_19590008_19591110_2 . . . 248
9.162 so_34906229_34906229_0 . . . 248
9.163 so_31398388_31398779_4 . . . 248
9.164 so_26754024_26754247_0 . . . 248
9.165 so_25566589_25566832_0 . . . 249
9.166 so_37897810_37897810_1 . . . 249
9.167 so_38215737_38226958_0 . . . 249
9.168 so_29858853_29858952_0 . . . 249
9.169 so_18818315_19146868_0 . . . 249
9.170 so_17757446_17761209_0 . . . 249
9.171 so_37132632_37132814_0 . . . 249
9.172 Class names . . . 249
9.173 so_2677221_14261176_0 . . . 250
9.174 so_4866425_4866845_0 . . . 250

9.175 so_31397071_31397245_2 . . . 250
9.176 so_2250872_2250898_0 . . . 250
9.177 so_5401575_5401792_0 . . . 250
9.178 so_3368129_3368191_3 . . . 250
9.179 so_33350002_33350762_1 . . . 250
9.180 so_33350002_33350762_0 . . . 250
9.181 so_40257011_40257321_2 . . . 250
9.182 so_32715610_32716199_1 . . . 250
9.183 so_1410563_1410632_0 . . . 250
9.184 so_30155802_30156450_1 . . . 250
9.185 so_6331394_6331423_0 . . . 250
9.186 so_24423481_24423481_0 . . . 250
9.187 so_22691821_22691915_0 . . . 250
9.188 so_38999076_39024593_0 . . . 250
9.189 so_5749892_5749947_0 . . . 250
9.190 so_4222651_4223081_0 . . . 251
9.191 so_11478962_11478962_5 . . . 251
9.192 so_15584561_15584643_0 . . . 251
9.193 so_13139235_13139291_0 . . . 251
9.194 Class members . . . 251
9.195 so_26561296_26561296_1 . . . 252
9.196 so_9789485_9790349_1 . . . 252
9.197 so_25354833_25355365_3 . . . 252
9.198 so_32712057_32713932_0 . . . 252
9.199 so_17957838_17958848_3 . . . 252
9.200 so_25300251_25300315_0 . . . 252
9.201 so_6301575_6301619_0 . . . 252
9.202 so_32715610_32716199_2 . . . 252
9.203 so_23712025_23712633_3 . . . 252
9.204 so_6419829_6419947_1 . . . 252
9.205 so_46952600_46952852_0 . . . 252
9.206 so_23625220_23625412_0 . . . 252
9.207 so_23625220_23625404_0 . . . 252
9.208 so_31224886_31225016_6 . . . 252
9.209 so_31022462_31022627_0 . . . 252
9.210 so_30047021_30048451_7 . . . 252
9.211 so_41148054_41148054_0 . . . 252
9.212 so_4884494_4885478_0 . . . 252
9.213 so_2421492_2421578_1 . . . 252
9.214 so_19469565_19469923_1 . . . 252
9.215 so_19469565_19469923_0 . . . 252
9.216 so_19469565_19480241_0 . . . 252
9.217 so_27699714_27702619_0 . . . 252
9.218 so_19529241_19532050_0 . . . 252
9.219 so_31713682_31713993_0 . . . 252
9.220 so_19590008_19591110_0 . . . 252
9.221 so_33462958_33463344_2 . . . 252
9.222 so_7046647_7047049_0 . . . 252
9.223 so_7046647_7046873_0 . . . 252
9.224 so_4296924_4297052_0 . . . 252
9.225 so_29302272_29302329_0 . . . 252

9.226 so_13484387_13484574_1 . . . 252
9.227 so_32685319_32686009_0 . . . 252
9.228 so_29196292_29197173_1 . . . 252
9.229 so_3601792_3601884_0 . . . 252
9.230 so_8422488_8422694_1 . . . 252
9.231 so_35790350_35790712_2 . . . 252
9.232 so_36587204_36607411_0 . . . 252
9.233 so_13225937_13226082_0 . . . 252
9.234 so_6331394_6331423_1 . . . 252
9.235 so_40368535_40369314_0 . . . 252
9.236 so_40368535_40369314_2 . . . 252
9.237 so_40368535_40369314_1 . . . 252
9.238 so_46446991_46446991_0 . . . 252
9.239 so_26435084_26435208_2 . . . 252
9.240 so_18847707_18855798_0 . . . 252
9.241 so_19564009_19565341_3 . . . 252
9.242 so_33907864_33907864_1 . . . 252
9.243 so_10924047_10924193_0 . . . 252
9.244 so_39381076_40426887_0 . . . 252
9.245 so_16524041_16524224_2 . . . 252
9.246 so_37194245_37194749_0 . . . 252
9.247 so_45653336_45655346_2 . . . 252
9.248 so_22507947_22508215_1 . . . 252
9.249 so_22507947_22508215_3 . . . 252
9.250 so_15478101_15479358_1 . . . 252
9.251 so_17585191_17585259_1 . . . 252
9.252 so_16573191_16573219_2 . . . 252
9.253 so_44044522_44045804_2 . . . 252
9.254 so_11438341_11438550_0 . . . 252
9.255 so_25822574_25822661_0 . . . 252
9.256 so_27754697_27754721_0 . . . 252
9.257 so_29412412_29412622_1 . . . 252
9.258 so_7538431_7538521_0 . . . 252
9.259 so_24545729_24545860_0 . . . 252
9.260 so_22544231_22544403_0 . . . 252
9.261 so_20832868_20833051_0 . . . 252
9.262 so_9241007_9241021_1 . . . 252
9.263 so_26614439_26614579_0 . . . 252
9.264 so_19564009_19565341_2 . . . 252
9.265 so_22507947_22508215_2 . . . 252
9.266 so_22098834_22098834_0 . . . 252
9.267 so_47455800_47455800_0 . . . 253
9.268 so_47263793_47268577_0 . . . 253
9.269 so_28526585_28527093_0 . . . 253
9.270 so_47592782_47592803_0 . . . 253
9.271 so_40590216_40590301_1 . . . 253
9.272 so_26939609_26939661_1 . . . 253
9.273 so_3140096_3140096_0 . . . 253
9.274 so_3824213_3824241_0 . . . 253
9.275 so_29841973_29841973_1 . . . 253
9.276 so_29841973_29841973_0 . . . 253

9.277 so_6577906_6578021_1 . . . 253
9.278 so_6577906_6578021_0 . . . 253
9.279 so_26408426_26408699_0 . . . 253
9.280 so_3838226_3838320_0 . . . 253
9.281 so_39026871_39026871_1 . . . 253
9.282 so_25479373_25479567_0 . . . 253
9.283 so_25479373_25479567_1 . . . 253
9.284 so_32700032_32700032_7 . . . 253
9.285 so_14414745_14414792_0 . . . 253
9.286 so_4438689_4438738_1 . . . 253
9.287 so_4978302_4979440_0 . . . 253
9.288 so_13585711_13586272_0 . . . 253
9.289 so_281045_281082_1 . . . 253
9.290 so_3276235_3276642_0 . . . 253
9.291 so_41407540_41407540_0 . . . 253
9.292 so_31082389_31082764_0 . . . 253
9.293 so_11340028_11340165_1 . . . 253
9.294 so_36256919_36257114_0 . . . 253
9.295 so_20706717_20706824_2 . . . 253
9.296 so_14352373_14352408_0 . . . 253
9.297 so_14352373_14352408_1 . . . 253
9.298 so_916600_916691_0 . . . 253
9.299 so_1876633_1876715_0 . . . 253
9.300 so_30599331_30599493_0 . . . 253
9.301 so_40991729_40992205_0 . . . 253
9.302 so_17627302_17628398_0 . . . 253
9.303 so_20653322_20653456_0 . . . 253
9.304 so_29280690_29280690_3 . . . 253
9.305 so_27953629_27953756_0 . . . 253
9.306 so_37373006_37373209_0 . . . 253
9.307 so_15463065_15544004_0 . . . 253
9.308 so_14186245_24445370_0 . . . 253
9.309 so_309922_309931_0 . . . 253
9.310 so_13792823_13792888_2 . . . 253
9.311 so_38650310_38701636_1 . . . 253
9.312 so_3454954_3455012_2 . . . 253
9.313 so_22824974_22825016_2 . . . 253
9.314 so_2624628_2624677_0 . . . 253
9.315 so_30168069_30168311_3 . . . 253
9.316 so_1632600_27682344_0 . . . 253
9.317 so_28526585_28527093_1 . . . 253
9.318 so_14361006_14361043_0 . . . 253
9.319 so_5829744_5830023_0 . . . 253
9.320 so_5829744_5830023_1 . . . 253
9.321 so_45332326_45332881_0 . . . 253
9.322 so_45332326_45332326_0 . . . 253
9.323 so_3820363_3820388_1 . . . 253
9.324 so_26529842_26529842_2 . . . 253
9.325 so_32106803_32106942_4 . . . 253
9.326 so_32106803_32106942_3 . . . 253
9.327 so_13861161_13862236_0 . . . 253

9.328 so_10788823_10789707_0 . . . 253
9.329 so_13541325_13543222_4 . . . 253
9.330 so_11494403_11494486_0 . . . 253
9.331 so_2881102_2881234_1 . . . 253
9.332 so_10448714_10449036_0 . . . 253
9.333 so_7762929_7762964_0 . . . 253
9.334 so_21499120_21499120_0 . . . 253
9.335 so_43988835_43989740_0 . . . 253
9.336 so_37323232_37369101_0 . . . 253
9.337 so_37373006_37373209_1 . . . 253
9.338 so_3454954_3455012_4 . . . 253
9.339 so_19843816_19868173_2 . . . 253
9.340 so_19590008_19591110_1 . . . 253
9.341 so_31553409_31557914_0 . . . 254
9.342 so_31553409_31557852_0 . . . 254
9.343 so_31553409_31557852_1 . . . 254
9.344 so_11373203_11906997_0 . . . 254
9.345 so_11373203_11906997_1 . . . 254
9.346 so_46323266_46323681_6 . . . 254
9.347 so_35333984_35336062_0 . . . 254
9.348 so_2310483_2310483_0 . . . 254
9.349 so_45341763_45344013_2 . . . 254
9.350 so_28521188_28521188_0 . . . 254
9.351 so_28521188_28522481_1 . . . 254
9.352 so_15277819_15277923_0 . . . 254
9.353 so_15277819_15278030_3 . . . 254
9.354 so_25542390_25543147_1 . . . 254
9.355 so_43988835_43989937_0 . . . 254
9.356 so_44523717_44524573_2 . . . 254
9.357 so_37373006_37373006_0 . . . 254
9.358 so_34616086_34616086_6 . . . 254
9.359 so_34616086_34616086_3 . . . 254
9.360 so_34616086_34616086_5 . . . 254
9.361 so_34616086_34616086_1 . . . 254
9.362 so_14357592_14357987_1 . . . 254
9.363 so_28786590_28787001_5 . . . 254
9.364 so_28786590_28787001_4 . . . 254
9.365 so_26939609_26939661_3 . . . 254
9.366 so_27137362_27138452_0 . . . 254
9.367 so_27137362_27139030_0 . . . 254
9.368 so_29277949_29278260_0 . . . 254
9.369 so_119123_37032302_1 . . . 254
9.370 so_119123_37032302_6 . . . 254
9.371 so_7157728_7157904_1 . . . 254
9.372 so_13484113_13484285_0 . . . 254
9.373 so_10788823_10788823_6 . . . 254
9.374 so_46323266_46323681_2 . . . 254
9.375 so_33069102_33277347_1 . . . 254
9.376 so_37005295_37005512_0 . . . 254
9.377 so_20832868_20833051_1 . . . 254
9.378 so_45557092_45557211_1 . . . 254

9.379 so_27615993_27616404_1 . . . 254
9.380 so_27615993_27616192_1 . . . 254
9.381 so_42447811_42447936_0 . . . 254
9.382 so_42447811_42447936_1 . . . 254
9.383 so_42860269_42860645_0 . . . 254
9.384 so_11340028_11340165_2 . . . 254
9.385 so_20706717_20706824_3 . . . 254
9.386 so_17845128_17845134_1 . . . 254
9.387 so_17845128_17845164_1 . . . 254
9.388 so_17845128_17845134_0 . . . 254
9.389 so_44409207_44416191_0 . . . 254
9.390 so_42420116_42420597_3 . . . 254
9.391 so_10448714_10449036_1 . . . 254
9.392 so_20653322_20653477_1 . . . 254
9.393 so_21499120_21509729_1 . . . 254
9.394 so_40476058_40476277_1 . . . 254
9.395 so_39599219_39599543_1 . . . 254
9.396 so_9254605_9254621_0 . . . 254
9.397 so_16962973_16966072_1 . . . 254
9.398 so_13792823_13792888_0 . . . 254
9.399 so_19786013_19786321_0 . . . 254
9.400 so_2624628_2624947_0 . . . 254
9.401 Member functions . . . 254
9.402 so_25299995_25300237_2 . . . 254
9.403 so_24083865_24083933_0 . . . 254
9.404 so_24083865_24083934_1 . . . 254
9.405 so_27007831_27008033_0 . . . 254
9.406 so_24229131_24229202_0 . . . 254
9.407 so_26055754_26055790_0 . . . 254
9.408 so_28183729_28183882_0 . . . 254
9.409 so_4945389_4945515_0 . . . 254
9.410 so_24345034_24345169_1 . . . 254
9.411 so_24345034_24345067_1 . . . 254
9.412 so_9474926_9474940_0 . . . 254
9.413 so_22003408_22003408_0 . . . 254
9.414 so_8392885_8393102_0 . . . 254
9.415 so_7518744_7518785_1 . . . 254
9.416 so_20550459_20550459_1 . . . 254
9.417 so_26921858_26922357_0 . . . 254
9.418 so_44692931_44693145_0 . . . 254
9.419 so_30030658_30030832_0 . . . 254
9.420 so_34989641_34989758_1 . . . 254
9.421 so_34989641_34989758_0 . . . 254
9.422 so_31812723_31813156_0 . . . 254
9.423 so_33463241_33463287_0 . . . 254
9.424 so_36841401_36841449_5 . . . 254
9.425 so_13375400_13375689_0 . . . 254
9.426 so_9734175_9734257_0 . . . 254
9.427 so_37638870_37640146_0 . . . 254
9.428 so_18104495_18105186_1 . . . 254
9.429 so_11420802_11470880_3 . . . 254

9.430 so_27608905_27609054_0 . . . 254
9.431 so_27608905_27609182_2 . . . 254
9.432 so_38929950_38931484_1 . . . 254
9.433 so_24345034_24345067_2 . . . 254
9.434 so_17536196_17536450_5 . . . 254
9.435 so_44788412_44788867_2 . . . 254
9.436 so_15865634_15865729_7 . . . 254
9.437 so_30030658_30030832_1 . . . 254
9.438 so_3571771_3571776_1 . . . 254
9.439 so_26821576_26821655_1 . . . 254
9.440 so_36841401_36841449_6 . . . 254
9.441 so_37638870_37640146_1 . . . 254
9.442 so_2152845_2152910_1 . . . 254
9.443 so_27608905_27609182_3 . . . 254
9.444 so_8203750_8203860_5 . . . 254
9.445 so_24345034_24345067_3 . . . 254
9.446 so_27608905_27609182_4 . . . 254
9.447 so_16493015_16498399_0 . . . 254
9.448 so_9474800_9474816_0 . . . 254
9.449 so_24293311_24313401_6 . . . 254
9.450 so_29825655_29825816_0 . . . 254
9.451 so_7119427_7119427_5 . . . 254
9.452 so_7119427_7120002_0 . . . 254
9.453 so_41667039_41667112_1 . . . 254
9.454 so_41667039_41667112_2 . . . 254
9.455 so_44994461_44994784_1 . . . 254
9.456 so_7111957_7112168_5 . . . 254
9.457 so_30354130_30356853_0 . . . 254
9.458 so_7173564_7173719_0 . . . 254
9.459 so_7173564_7173719_1 . . . 254
9.460 so_42457227_42457407_0 . . . 255
9.461 so_36821049_36821822_0 . . . 255
9.462 so_37115503_37116610_0 . . . 255
9.463 so_18979503_18979812_0 . . . 255
9.464 so_1025345_1950150_0 . . . 255
9.465 so_29455791_29455791_0 . . . 255
9.466 so_14217746_14218744_0 . . . 255
9.467 so_40249854_40250232_0 . . . 255
9.468 so_26667756_26667756_0 . . . 255
9.469 so_26687808_26687847_0 . . . 255
9.470 so_47120674_47120833_0 . . . 255
9.471 so_25091602_25092726_1 . . . 255
9.472 so_25741239_25741602_0 . . . 255
9.473 so_2474018_2474021_3 . . . 255
9.474 so_40530874_40538421_0 . . . 255
9.475 so_26341160_26344975_0 . . . 255
9.476 so_41352132_41352253_2 . . . 255
9.477 so_10278199_10278199_1 . . . 255
9.478 so_25974056_25974400_0 . . . 255
9.479 so_29990849_29990901_0 . . . 255
9.480 so_28118360_28118452_0 . . . 255

9.481 so_28118360_28118788_0 . . . 255
9.482 so_28118360_28118360_0 . . . 255
9.483 so_35496609_35497520_1 . . . 255
9.484 so_27722358_27732038_1 . . . 255
9.485 so_46646773_46647140_0 . . . 255
9.486 so_25376209_25376509_0 . . . 255
9.487 so_17642624_17648572_0 . . . 255
9.488 so_13162465_13162653_1 . . . 255
9.489 so_38195245_38195986_5 . . . 255
9.490 so_24500386_24500631_0 . . . 255
9.491 so_43936404_43937134_2 . . . 255
9.492 so_7210496_7210512_6 . . . 255
9.493 so_7049528_7049548_1 . . . 255
9.494 so_12656548_12656790_1 . . . 255
9.495 so_16524041_16524361_0 . . . 255
9.496 so_16524897_16525136_1 . . . 255
9.497 so_16524897_16525136_0 . . . 255
9.498 so_25321859_25321947_0 . . . 255
9.499 so_22419578_22419598_3 . . . 256
9.500 so_40796662_40967442_0 . . . 256
9.501 so_11970707_11970755_0 . . . 256
9.502 so_8576509_8576705_0 . . . 256
9.503 so_8952593_8952728_3 . . . 256
9.504 so_27340009_27340438_0 . . . 256
9.505 so_16389641_16389649_0 . . . 256
9.506 so_16178061_16178093_3 . . . 256
9.507 so_3330620_3330652_0 . . . 256
9.508 so_21861148_21861190_0 . . . 256
9.509 so_4059932_4059936_1 . . . 256
9.510 so_4059932_4059936_0 . . . 256
9.511 so_28346426_28346853_0 . . . 256
9.512 so_20021292_20021356_4 . . . 256
9.513 so_13842682_13843053_1 . . . 256
9.514 so_13842682_13843022_0 . . . 256
9.515 so_23443442_23443514_0 . . . 256
9.516 so_15624145_15624203_0 . . . 256
9.517 so_27400267_27400667_0 . . . 256
9.518 so_18126737_18127358_1 . . . 256
9.519 so_24762862_24766972_0 . . . 256
9.520 so_13476879_13476889_0 . . . 256
9.521 so_645994_3899282_1 . . . 256
9.522 so_31362252_31362252_0 . . . 256
9.523 so_11333671_11333671_0 . . . 256
9.524 so_2726146_2726159_0 . . . 256
9.525 so_44062328_44063793_0 . . . 256
9.526 so_8576509_8576705_1 . . . 256
9.527 so_20083516_20083551_0 . . . 256
9.528 so_41131299_41131420_1 . . . 256
9.529 so_33477090_33477809_0 . . . 256
9.530 so_3484233_3484265_1 . . . 256
9.531 so_3484233_3484265_2 . . . 256

9.532 so_27606364_27606511_0 . . . 256
9.533 so_4520900_4521571_0 . . . 256
9.534 so_18292834_18293358_1 . . . 256
9.535 so_2444734_2444902_0 . . . 256
9.536 so_15279508_15280016_0 . . . 256
9.537 so_43936404_43936807_1 . . . 256
9.538 so_43936404_43937134_1 . . . 256
9.539 so_35229620_35230278_0 . . . 256
9.540 so_18868004_18868236_1 . . . 256
9.541 so_16178061_16178094_0 . . . 256
9.542 so_3028632_8621112_0 . . . 256
9.543 so_24423480_24423563_0 . . . 256
9.544 so_3330620_3330652_1 . . . 256
9.545 so_8610571_8610728_13 . . . 256
9.546 so_5365689_5365714_11 . . . 256
9.547 so_30258639_30259298_0 . . . 256
9.548 so_16636563_16636599_0 . . . 256
9.549 so_33679419_33679487_0 . . . 256
9.550 so_16585562_16592164_0 . . . 256
9.551 so_6067244_6067267_1 . . . 256
9.552 so_4532866_4537163_0 . . . 256
9.553 so_46330784_46331111_0 . . . 256
9.554 so_4826719_4826944_0 . . . 257
9.555 so_30512204_30512259_0 . . . 257
9.556 so_41851079_41851079_0 . . . 257
9.557 so_27340009_27340438_1 . . . 257
9.558 so_16775787_16775812_0 . . . 257
9.559 Static members . . . 257
9.560 so_12151127_12151232_2 . . . 257
9.561 so_46930702_46931313_0 . . . 257
9.562 so_11977137_11977222_0 . . . 257
9.563 so_8642219_8642366_0 . . . 257
9.564 so_8642219_8642338_0 . . . 257
9.565 so_3498444_3498473_0 . . . 257
9.566 so_10157296_10157328_0 . . . 257
9.567 so_23756459_23771621_0 . . . 257
9.568 so_44995196_44996066_0 . . . 257
9.569 so_325555_325572_0 . . . 257
9.570 so_2474018_2474021_4 . . . 257
9.571 so_46544292_46544392_0 . . . 257
9.572 so_2207320_2207499_0 . . . 257
9.573 so_28482809_28483254_1 . . . 257
9.574 so_5365689_5365714_7 . . . 257
9.575 so_12151127_12151232_3 . . . 257
9.576 so_41263504_41454584_0 . . . 257
9.577 so_41263504_41263504_1 . . . 257
9.578 so_5830060_5830131_0 . . . 258
9.579 so_36985082_36985599_0 . . . 258
9.580 so_12151127_12151232_4 . . . 258
9.581 so_5830060_5830131_1 . . . 258
9.582 so_19266935_19267247_0 . . . 258

9.583 so_34419603_34419603_0 . . . 258
9.584 so_26530782_26530842_0 . . . 258
9.585 so_27583384_27583402_0 . . . 258
9.586 so_19812786_19812883_0 . . . 258
9.587 so_19812786_19812883_1 . . . 258
9.588 so_18623111_18623401_1 . . . 258
9.589 so_42332147_42332651_0 . . . 258
9.590 so_42332147_42332886_1 . . . 258
9.591 so_11970707_11970755_1 . . . 258
9.592 so_12997967_12997997_0 . . . 258
9.593 so_16178061_16178093_2 . . . 258
9.594 so_34997194_34997272_1 . . . 258
9.595 so_27317838_27317838_1 . . . 258
9.596 so_16434614_16434677_0 . . . 258
9.597 so_6362324_6362427_0 . . . 258
9.598 so_4640989_4640999_0 . . . 258
9.599 so_4842091_4842128_0 . . . 258
9.600 so_3792412_3792427_0 . . . 258
9.601 so_19547504_19547526_0 . . . 258
9.602 so_33908822_33910325_3 . . . 258
9.603 so_43936404_43937134_0 . . . 258
9.604 so_4842056_4842120_0 . . . 258
9.605 so_2894429_2894505_0 . . . 258
9.606 so_3673831_3674214_0 . . . 258
9.607 so_41263504_41263504_2 . . . 258
9.608 so_24865741_24865832_0 . . . 258
9.609 so_46681235_46682497_0 . . . 258
9.610 so_26923781_26924129_5 . . . 258
9.611 so_23895081_23895527_5 . . . 258
9.612 so_47311823_47312173_0 . . . 258
9.613 so_47311823_47312408_0 . . . 258
9.614 so_45081096_45081406_0 . . . 258
9.615 so_5032683_5032811_0 . . . 258
9.616 so_46874055_46874207_0 . . . 258
9.617 so_29432283_29432462_1 . . . 258
9.618 so_13347116_13347587_0 . . . 258
9.619 so_30929941_30929971_0 . . . 258
9.620 so_4241946_4242093_0 . . . 258
9.621 so_4104544_4104649_0 . . . 258
9.622 so_630946_631195_1 . . . 258
9.623 so_17432891_17432891_0 . . . 258
9.624 so_26648817_26649961_15 . . . 258
9.625 so_4640989_4640999_1 . . . 258
9.626 so_26870810_26871714_1 . . . 258
9.627 so_26870810_26871714_0 . . . 258
9.628 so_16300432_16300433_1 . . . 258
9.629 so_16300432_16302929_4 . . . 258
9.630 so_33074110_33074895_0 . . . 258
9.631 so_29954191_29954631_0 . . . 258
9.632 so_19266935_19267036_0 . . . 258
9.633 so_4083971_4084003_0 . . . 258

9.634 so_19341360_19341402_0 . . . 258
9.635 so_36440927_36440966_1 . . . 258
9.636 so_36440927_36440966_0 . . . 258
9.637 so_16049306_16049365_0 . . . 258
9.638 so_3536372_3536498_3 . . . 258
9.639 so_20599618_20599758_0 . . . 258
9.640 so_13442909_13442926_0 . . . 258
9.641 so_19344676_19344738_0 . . . 258
9.642 so_14395967_14395967_1 . . . 258
9.643 so_19267626_19267667_1 . . . 258
9.644 so_5359930_5359954_0 . . . 258
9.645 so_41953456_41953794_1 . . . 258
9.646 so_24273136_24273136_0 . . . 258
9.647 so_23767241_23767278_1 . . . 258
9.648 so_24237729_24252540_0 . . . 258
9.649 so_34261861_34262158_0 . . . 258
9.650 so_16730336_16730438_0 . . . 258
9.651 so_13302425_13302614_0 . . . 258
9.652 so_9241007_9241021_2 . . . 258
9.653 so_23983916_23984111_2 . . . 258
9.654 so_23983916_23984128_1 . . . 258
9.655 so_23926678_23926951_0 . . . 258
9.656 so_23926678_23926951_1 . . . 258
9.657 so_3792412_3792427_1 . . . 258
9.658 so_19547504_19547526_1 . . . 258
9.659 so_7258363_7258467_0 . . . 258
9.660 so_33908822_33910325_5 . . . 258
9.661 so_33908822_33910325_4 . . . 258
9.662 so_33908822_33910325_6 . . . 258
9.663 so_3025997_3026237_0 . . . 258
9.664 so_22535154_22535349_0 . . . 258
9.665 so_16373831_16373831_0 . . . 258
9.666 so_578654_578693_2 . . . 258
9.667 so_3698446_3698475_0 . . . 258
9.668 so_41148054_41195598_3 . . . 258
9.669 so_38583989_38584106_2 . . . 258
9.670 so_38583989_38584106_0 . . . 258
9.671 so_29799766_29800268_0 . . . 258
9.672 so_27581077_29921347_0 . . . 258
9.673 so_4842056_4842120_1 . . . 258
9.674 so_24682349_24682473_1 . . . 258
9.675 so_28134744_28134991_0 . . . 258
9.676 so_16258601_16258962_1 . . . 258
9.677 so_19744917_19744977_1 . . . 258
9.678 so_27699714_27699714_0 . . . 258
9.679 so_2894429_2894505_1 . . . 258
9.680 so_3673831_3674214_1 . . . 258
9.681 so_19733992_19734092_0 . . . 258
9.682 so_19004624_19004670_0 . . . 258
9.683 so_29976770_29976855_0 . . . 258
9.684 so_20700843_20701040_0 . . . 258

9.685 so_22507947_22508215_4 . . . 258
9.686 so_43197543_43197624_1 . . . 258
9.687 so_5391973_5392050_0 . . . 259
9.688 so_8016780_28846608_0 . . . 259
9.689 so_31565836_31565836_0 . . . 259
9.690 so_46510358_46510386_0 . . . 259
9.691 so_35792937_35834427_0 . . . 259
9.692 so_712334_712534_1 . . . 259
9.693 so_5032683_5032811_1 . . . 259
9.694 so_1907214_1907353_0 . . . 259
9.695 so_1907214_1907283_2 . . . 259
9.696 so_26196095_28446388_3 . . . 259
9.697 so_9656941_9657064_4 . . . 259
9.698 so_9656941_9657064_0 . . . 259
9.699 so_24018932_24020013_1 . . . 259
9.700 so_9219898_9219987_1 . . . 259
9.701 so_9219898_9219987_0 . . . 259
9.702 so_29432283_29432462_0 . . . 259
9.703 so_13347116_13347587_1 . . . 259
9.704 so_44871322_44871623_0 . . . 259
9.705 so_630946_631195_2 . . . 259
9.706 so_11336173_11336240_0 . . . 259
9.707 so_26648817_26649961_1 . . . 259
9.708 so_26648817_26649961_0 . . . 259
9.709 so_34792403_34792443_0 . . . 259
9.710 so_21072061_21072181_0 . . . 259
9.711 so_33204020_33204136_0 . . . 259
9.712 so_13571749_13784100_4 . . . 259
9.713 so_14547986_14548934_0 . . . 259
9.714 so_11300652_11301021_0 . . . 259
9.715 so_3531296_3531395_0 . . . 259
9.716 so_4083971_4083993_0 . . . 259
9.717 so_4083971_4084003_1 . . . 259
9.718 so_6106194_6106240_0 . . . 259
9.719 so_1257267_1257366_0 . . . 259
9.720 so_28324686_28324791_0 . . . 259
9.721 so_30742288_30742473_2 . . . 259
9.722 so_30742288_30742473_3 . . . 259
9.723 so_28530583_28535343_1 . . . 259
9.724 so_29067279_29067638_1 . . . 259
9.725 so_25573711_25573937_0 . . . 259
9.726 so_32433554_32433923_0 . . . 259
9.727 so_32433554_32433916_0 . . . 259
9.728 so_11178434_11178565_0 . . . 259
9.729 so_29844028_29844272_0 . . . 259
9.730 so_35140997_35142328_0 . . . 259
9.731 so_9213985_9214528_1 . . . 259
9.732 so_9213985_9214528_0 . . . 259
9.733 so_2454019_2454082_1 . . . 259
9.734 so_2454019_2454132_0 . . . 259
9.735 so_28347008_28353658_0 . . . 259

9.736 so_5151683_5151723_0 . . . 259
9.737 so_2738435_2738576_0 . . . 259
9.738 so_2738435_2738576_1 . . . 259
9.739 so_14395967_14396189_0 . . . 259
9.740 so_40018653_40020448_0 . . . 259
9.741 so_4891067_4891102_2 . . . 259
9.742 so_4891067_4891102_1 . . . 259
9.743 so_44721729_44725331_1 . . . 259
9.744 so_44721729_44725331_0 . . . 259
9.745 so_4547660_4548031_0 . . . 259
9.746 so_31074546_31074546_0 . . . 259
9.747 so_42799268_42799405_0 . . . 259
9.748 so_27258344_27258829_0 . . . 259
9.749 so_45190999_45192353_1 . . . 259
9.750 so_23767241_23767278_2 . . . 259
9.751 so_16730336_16730439_0 . . . 259
9.752 so_16730336_16730438_1 . . . 259
9.753 so_24572428_24573119_0 . . . 259
9.754 so_23250651_23250909_4 . . . 259
9.755 so_28264279_28462742_0 . . . 259
9.756 so_12859831_12860580_0 . . . 259
9.757 so_3792412_3792427_2 . . . 259
9.758 so_31860300_31860548_0 . . . 259
9.759 so_4209433_4209595_0 . . . 259
9.760 so_21344031_21344184_0 . . . 259
9.761 so_34520851_34551358_0 . . . 259
9.762 so_18107077_18107129_1 . . . 259
9.763 so_11928089_11929560_0 . . . 259
9.764 so_5183703_5183838_0 . . . 259
9.765 so_34053606_34053854_0 . . . 259
9.766 so_24708952_24709095_0 . . . 259
9.767 so_24708952_24709072_0 . . . 259
9.768 so_16373831_16374286_0 . . . 259
9.769 so_6654609_6654820_0 . . . 259
9.770 so_42516175_42516175_2 . . . 259
9.771 so_16190776_16190777_1 . . . 259
9.772 so_16190776_16190777_0 . . . 259
9.773 so_43172652_43174512_2 . . . 259
9.774 so_37816186_37818272_2 . . . 259
9.775 so_37816186_37818272_3 . . . 259
9.776 so_16467802_16467846_0 . . . 259
9.777 so_30831367_30831411_0 . . . 259
9.778 so_19744917_19744977_2 . . . 259
9.779 so_8162994_8163317_0 . . . 259
9.780 so_17074385_17074933_0 . . . 259
9.781 so_44739864_44740285_2 . . . 259
9.782 so_44739864_44740285_1 . . . 259
9.783 so_45719407_45719751_1 . . . 259
9.784 so_11522399_11800201_0 . . . 259
9.785 so_11522399_11523155_0 . . . 259
9.786 so_18087464_18087591_0 . . . 259

9.787 so_13390376_13390376_2 . . . 259
9.788 so_12489597_12489697_0 . . . 259
9.789 so_1639154_1639164_0 . . . 259
9.790 so_1639154_1639194_0 . . . 259
9.791 so_34649138_34649138_0 . . . 259
9.792 so_45719407_45719751_2 . . . 259
9.793 so_26923781_26924129_2 . . . 259
9.794 so_6415509_6415621_3 . . . 259
9.795 so_15443901_15443968_0 . . . 259
9.796 so_23254270_23254270_1 . . . 259
9.797 so_3698043_3698139_0 . . . 259
9.798 so_8162994_8162994_0 . . . 259
9.799 so_5714616_5714720_4 . . . 259
9.800 so_26777821_35142255_0 . . . 259
9.801 so_8203750_8203860_4 . . . 259
9.802 so_6939020_6939059_0 . . . 259
9.803 so_5658499_5658636_0 . . . 259
9.804 so_4640989_4640999_2 . . . 259
9.805 so_16300432_16300433_3 . . . 259
9.806 so_16300432_16300433_2 . . . 259
9.807 so_2204608_2204628_1 . . . 259
9.808 so_5359930_5359954_1 . . . 259
9.809 so_1255460_1255659_0 . . . 259
9.810 so_4842056_4842120_2 . . . 259
9.811 so_2894429_2894505_2 . . . 259
9.812 so_4699084_4699110_0 . . . 259
9.813 so_24531186_24531655_1 . . . 259
9.814 so_1639154_1639194_1 . . . 259
9.815 Unions . . . 259
9.816 so_33921710_36993074_4 . . . 259
9.817 so_23004941_23005015_3 . . . 259
9.818 so_38058482_38059960_1 . . . 259
9.819 so_25664848_25672839_3 . . . 259
9.820 so_39757246_39757263_0 . . . 259
9.821 so_31388208_31388208_0 . . . 259
9.822 so_6136010_6136130_0 . . . 259
9.823 so_10271929_10272008_0 . . . 259
9.824 so_42361656_42367505_3 . . . 259
9.825 so_42361656_42367505_1 . . . 259
9.826 so_11373203_11996970_3 . . . 259
9.827 so_11373203_11373277_1 . . . 259
9.828 so_2209571_2209697_0 . . . 259
9.829 so_20806185_20806185_1 . . . 259
9.830 so_5246389_5246510_0 . . . 259
9.831 so_33069102_33277347_2 . . . 259
9.832 so_33056403_33056605_3 . . . 259
9.833 so_33056403_33056605_2 . . . 259
9.834 so_21763002_21763025_0 . . . 259
9.835 so_22022826_22026484_0 . . . 259
9.836 so_34677343_34677577_2 . . . 259
9.837 so_34677343_34677577_1 . . . 259

9.838 so_28521188_28528989_0 . . . 259
9.839 so_39763548_39763548_0 . . . 259
9.840 so_39763548_39780208_0 . . . 259
9.841 so_2881102_2881234_0 . . . 259
9.842 so_42420116_42420597_2 . . . 259
9.843 so_19136881_19137295_1 . . . 259
9.844 so_29280130_29280130_0 . . . 259
9.845 so_16782746_16782797_0 . . . 259
9.846 so_740577_740686_0 . . . 259
9.847 so_26560311_26560311_0 . . . 259
9.848 so_33058717_33058919_0 . . . 259
9.849 so_33058717_33058919_1 . . . 259
9.850 so_18128648_18168460_2 . . . 259
9.851 so_41487696_41488396_1 . . . 259
9.852 so_14357592_14357987_0 . . . 259
9.853 so_16637074_16637074_0 . . . 259
9.854 so_32738518_32739887_0 . . . 259
9.855 so_23005241_23005241_0 . . . 259
9.856 so_17790278_17790901_0 . . . 259
9.857 so_33921710_36993074_3 . . . 260
9.858 so_31026316_31026693_0 . . . 260
9.859 so_26572240_26572240_0 . . . 260
9.860 so_39745566_39750232_0 . . . 260
9.861 so_34966340_34966340_2 . . . 260
9.862 so_19764150_19764531_0 . . . 260
9.863 so_3521914_3521998_1 . . . 260
9.864 so_38691282_38691329_0 . . . 260
9.865 so_20805215_20806004_2 . . . 260
9.866 so_28063948_28064049_0 . . . 260
9.867 so_28334899_28336094_0 . . . 260
9.868 so_3615001_3615001_1 . . . 260
9.869 so_3615001_3615001_0 . . . 260
9.870 so_30417121_30417151_1 . . . 260
9.871 so_1873219_1873233_0 . . . 260
9.872 so_6014717_6014887_1 . . . 260
9.873 so_10781130_10781727_0 . . . 260
9.874 so_46402983_46403511_0 . . . 260
9.875 so_22780009_22783981_0 . . . 260
9.876 so_16662939_16663114_0 . . . 260
9.877 so_4397019_4397125_0 . . . 260
9.878 so_34581372_34581372_2 . . . 260
9.879 so_30492927_30493617_0 . . . 260
9.880 so_40106941_40107282_0 . . . 260
9.881 so_8837835_8837889_0 . . . 260
9.882 so_4773334_4773351_0 . . . 260
9.883 so_5275100_5275191_0 . . . 260
9.884 so_25778072_25778139_0 . . . 260
9.885 so_10693913_10693913_0 . . . 260
9.886 so_22780009_22783981_1 . . . 260
9.887 so_16662939_16663114_2 . . . 260
9.888 so_17923783_17923935_0 . . . 260

9.889 so_23004941_23005015_1 . . . 260
9.890 so_23004941_23005015_0 . . . 260
9.891 so_31388208_31388208_1 . . . 260
9.892 so_47168371_47168371_0 . . . 260
9.893 so_19764150_19764531_1 . . . 260
9.894 so_9546914_9547181_0 . . . 260
9.895 so_24713833_24714262_0 . . . 260
9.896 so_22022826_22026484_2 . . . 260
9.897 so_33936295_33937081_0 . . . 260
9.898 so_31049885_31050495_1 . . . 260
9.899 so_39763548_39764671_4 . . . 260
9.900 so_37373006_37373209_4 . . . 260
9.901 so_32641531_32642696_3 . . . 260
9.902 so_14248044_14248134_1 . . . 260
9.903 so_14248044_14248134_2 . . . 260
9.904 so_5453170_5453325_0 . . . 260
9.905 so_4074651_4074813_0 . . . 260
9.906 so_25542390_25543147_0 . . . 260
9.907 so_13624760_35388572_0 . . . 260
9.908 so_13624760_13624921_0 . . . 260
9.909 so_2253878_2253971_1 . . . 260
9.910 so_36393588_36393902_1 . . . 260
9.911 so_46056697_46056836_0 . . . 260
9.912 so_4074651_4074651_0 . . . 261
9.913 so_10693913_10694924_0 . . . 261
9.914 so_21093669_21093800_0 . . . 261
9.915 so_26063812_26063851_0 . . . 261
9.916 so_28636352_28636352_0 . . . 261
9.917 so_42416258_42418091_2 . . . 261
9.918 so_25778072_25778194_0 . . . 261
9.919 so_25778072_25778072_0 . . . 261
9.920 so_43565457_43565560_0 . . . 261
9.921 Bit-fields . . . 261
9.922 so_28786590_28787001_1 . . . 261
9.923 so_28786590_28787001_2 . . . 261
9.924 so_13114752_13115617_0 . . . 261
9.925 so_2647320_2647442_0 . . . 261
9.926 so_2647320_2648520_1 . . . 261
9.927 so_25479373_25479567_2 . . . 261
9.928 so_39415528_39415639_0 . . . 261
9.929 so_27506462_27506551_3 . . . 261
9.930 so_13810222_13810408_2 . . . 261
9.931 so_38335882_38336704_1 . . . 261
9.932 so_31726191_31727805_0 . . . 261
9.933 so_43333327_43365697_0 . . . 261
9.934 so_26998571_26999748_3 . . . 261
9.935 so_22995226_22995621_0 . . . 261
9.936 so_38137625_39549779_1 . . . 261
9.937 so_43193283_43193283_0 . . . 261
9.938 so_40655160_40655299_0 . . . 261
9.939 so_47193195_47193591_1 . . . 262

9.940 so_13810222_13810408_3 . . . 262
9.941 so_13810222_13810408_16 . . . 262
9.942 so_27115167_27115207_2 . . . 262
9.943 so_4297095_4297110_0 . . . 262
9.944 so_25479373_25479567_3 . . . 262
9.945 so_30019454_30019483_0 . . . 262
9.946 so_13171054_13171148_0 . . . 262
9.947 so_32691535_32691642_0 . . . 262
9.948 so_13810222_13810408_4 . . . 262
9.949 so_43085359_43085658_0 . . . 262
9.950 so_11100443_11100685_1 . . . 262
9.951 so_11100443_11100497_0 . . . 262
9.952 so_14026202_14026224_0 . . . 262
9.953 so_33723631_33725514_0 . . . 262
9.954 so_33723631_33725514_1 . . . 262
9.955 so_13171054_13171154_1 . . . 262
9.956 so_13810222_13810408_5 . . . 262
9.957 so_29855708_29864781_0 . . . 262
9.958 so_11983231_11983423_1 . . . 262
9.959 so_25940635_25940635_1 . . . 262
9.960 Nested class declarations . . . 262
9.961 so_45650554_45650950_1 . . . 262
9.962 so_7878124_7878481_1 . . . 262
9.963 so_12063056_12063166_0 . . . 262
9.964 so_2421492_2421590_0 . . . 262
9.965 so_23865908_23866108_0 . . . 262
9.966 so_45650554_45650950_2 . . . 263
9.967 so_38464658_38464658_0 . . . 263
9.968 so_32595213_32595306_1 . . . 263
9.969 so_45650554_45650950_3 . . . 263
9.970 so_31659851_31660281_1 . . . 263
9.971 so_31504163_31504163_0 . . . 263
9.972 so_45650554_45650950_4 . . . 263
9.973 so_3537969_3537998_0 . . . 263
9.974 so_2881305_2881463_0 . . . 263
9.975 Local class declarations . . . 263
9.976 so_14801591_14804071_1 . . . 263
9.977 so_6415509_6415621_0 . . . 263
9.978 so_41724428_41724428_2 . . . 263
9.979 so_41724428_41724428_1 . . . 263
9.980 so_18491341_18491519_1 . . . 263
9.981 so_7878124_7878481_3 . . . 263
9.982 so_3930204_3930239_0 . . . 263
9.983 so_23834845_23835017_0 . . . 263
9.984 so_23834845_23835015_1 . . . 263
9.985 so_5714616_5714720_1 . . . 263
9.986 so_6939020_6939059_1 . . . 263
9.987 so_6415509_6415621_1 . . . 264
9.988 so_46873142_46873181_0 . . . 264
9.989 so_5714616_5714720_2 . . . 264
9.990 so_6415509_6415621_2 . . . 264

9.991 so_5714616_5714720_3 . . . 264
9.992 Nested type names . . . 264
9.993 so_37206557_37207250_0 . . . 264
9.994 so_23712025_23712633_0 . . . 264
9.995 so_2952216_2952216_0 . . . 264
9.996 so_2421492_2421590_2 . . . 264

10 Derived classes **265**
10.1 so_3849724_3849776_0 . . . 265
10.2 so_41033384_41033666_0 . . . 265
10.3 so_29362582_29363199_1 . . . 265
10.4 so_5813499_5813499_0 . . . 265
10.5 so_4305597_4305758_5 . . . 265
10.6 so_2569782_2569819_0 . . . 265
10.7 so_7420314_7420417_0 . . . 265
10.8 so_8422488_8422694_0 . . . 265
10.9 so_8422488_8423121_0 . . . 265
10.10 so_43596484_43596754_0 . . . 265
10.11 so_47489881_47490236_0 . . . 265
10.12 so_15090216_15090354_1 . . . 265
10.13 so_31361306_31361306_5 . . . 265
10.14 so_31361306_31361336_0 . . . 265
10.15 so_39904101_39904258_1 . . . 265
10.16 so_19529241_19532050_1 . . . 265
10.17 so_7332279_7332866_0 . . . 265
10.18 so_4347730_4347986_1 . . . 265
10.19 so_3849724_3849776_1 . . . 265
10.20 so_39384399_39385357_1 . . . 265
10.21 so_39384399_39385357_2 . . . 265
10.22 so_42447811_42447936_3 . . . 266
10.23 so_3902513_3902513_4 . . . 266
10.24 so_39791806_39866710_0 . . . 266
10.25 so_42501878_42503334_1 . . . 266
10.26 so_7113422_7114379_0 . . . 266
10.27 so_13734282_13734366_0 . . . 266
10.28 so_23498746_23502492_1 . . . 266
10.29 so_23498746_23499592_1 . . . 266
10.30 so_30826000_30826000_0 . . . 266
10.31 Multiple base classes . . . 266
10.32 so_20405321_20405376_0 . . . 266
10.33 so_17328921_17328962_1 . . . 266
10.34 so_17328921_17329506_0 . . . 266
10.35 so_16185945_16186132_0 . . . 266
10.36 so_31518581_31518644_0 . . . 266
10.37 so_18402753_18402840_0 . . . 267
10.38 so_19986123_19986123_0 . . . 267
10.39 so_5910301_5910402_0 . . . 267
10.40 so_3608108_3608208_0 . . . 267
10.41 so_10353053_10353511_0 . . . 267
10.42 so_18402753_18402840_1 . . . 267
10.43 so_42979185_42979681_0 . . . 267

10.44 so_16886788_16889408_2 . . . 267
10.45 so_16886788_16886788_0 . . . 267
10.46 so_16886788_16886788_1 . . . 267
10.47 so_31006916_31007045_0 . . . 267
10.48 so_25324582_25325041_0 . . . 267
10.49 so_10353053_10353511_1 . . . 267
10.50 so_18402753_18402840_2 . . . 267
10.51 so_36648411_36648698_2 . . . 267
10.52 Member name lookup . . . 268
10.53 so_32566579_32567205_0 . . . 268
10.54 so_39042240_39046515_1 . . . 268
10.55 so_21636150_21637519_0 . . . 268
10.56 so_34671852_34671852_0 . . . 268
10.57 so_19947927_19948361_0 . . . 268
10.58 so_43596484_43596754_1 . . . 268
10.59 so_25066001_25066055_1 . . . 268
10.60 so_28442484_28442828_1 . . . 268
10.61 so_24351583_24351606_1 . . . 268
10.62 so_24299963_24299963_1 . . . 269
10.63 so_34671852_34671852_1 . . . 269
10.64 so_9117603_9117779_0 . . . 269
10.65 so_24351583_24351606_2 . . . 269
10.66 so_28348429_28349054_1 . . . 269
10.67 so_31525445_31526379_1 . . . 269
10.68 so_44626415_44720780_0 . . . 269
10.69 so_34671852_34671852_2 . . . 269
10.70 so_34671852_34671852_4 . . . 269
10.71 so_33694192_33696684_0 . . . 269
10.72 so_43596484_43596754_2 . . . 269
10.73 so_19340190_19340231_2 . . . 269
10.74 so_35373384_35726062_1 . . . 269
10.75 so_30036255_30036657_0 . . . 269
10.76 so_17642022_17646210_0 . . . 269
10.77 so_9117603_9117779_1 . . . 269
10.78 so_24351583_24351606_3 . . . 269
10.79 so_24299963_24299963_4 . . . 269
10.80 so_29595028_29595219_0 . . . 269
10.81 so_4094253_4094253_0 . . . 269
10.82 so_40608180_40608703_1 . . . 269
10.83 so_34671852_34671852_10 . . . 269
10.84 so_34671852_34671852_3 . . . 269
10.85 so_9117603_9117779_2 . . . 269
10.86 so_24351583_24351606_4 . . . 269
10.87 so_28348429_28349054_0 . . . 269
10.88 so_31525445_31526379_3 . . . 269
10.89 so_24095412_24097243_2 . . . 269
10.90 so_24095412_24097243_1 . . . 269
10.91 so_35373384_35726062_2 . . . 269
10.92 so_29661253_29661678_0 . . . 269
10.93 so_23619121_23619395_0 . . . 269
10.94 so_9117603_9117779_3 . . . 269

10.95 so_24351583_24351606_5 . . . 269
10.96 so_31525445_31526379_4 . . . 269
10.97 so_35373384_35726062_3 . . . 269
10.98 so_44429408_44436678_3 . . . 269
10.99 so_44429408_44436678_0 . . . 269
10.100 so_44429408_44436678_2 . . . 269
10.101 so_44429408_44436678_1 . . . 269
10.102 so_9117603_9117779_4 . . . 269
10.103 so_19276238_19277394_1 . . . 269
10.104 so_15398558_15421410_2 . . . 269
10.105 so_35373384_35726062_4 . . . 269
10.106 so_39042240_39046515_2 . . . 270
10.107 so_21636150_21637519_1 . . . 270
10.108 so_43596484_43596754_3 . . . 270
10.109 so_15398558_15421410_3 . . . 270
10.110 so_37206557_37207250_1 . . . 270
10.111 so_44626415_44720780_2 . . . 270
10.112 so_4742737_4742737_0 . . . 271
10.113 Virtual functions . . . 272
10.114 so_23945043_23945179_1 . . . 272
10.115 so_24920218_24920218_2 . . . 272
10.116 so_6796650_6796773_0 . . . 272
10.117 so_6796650_6796802_0 . . . 272
10.118 so_23518993_23543949_0 . . . 272
10.119 so_31683281_31683410_0 . . . 272
10.120 so_5831863_5832682_0 . . . 272
10.121 so_7236417_7236574_0 . . . 272
10.122 so_35422885_35423041_0 . . . 272
10.123 so_35422885_35423114_0 . . . 272
10.124 so_4227328_4227378_3 . . . 272
10.125 so_18147864_18149769_2 . . . 272
10.126 so_7687041_7688154_0 . . . 272
10.127 so_31708731_31709049_1 . . . 272
10.128 so_30785257_30785910_1 . . . 272
10.129 so_30785257_30785910_0 . . . 272
10.130 so_1404797_1404843_1 . . . 272
10.131 so_1404797_1404843_0 . . . 272
10.132 so_24728093_24728123_0 . . . 272
10.133 so_39162226_39162297_2 . . . 272
10.134 so_39162226_39162297_1 . . . 272
10.135 so_4241946_4242093_2 . . . 272
10.136 so_36665029_36665206_0 . . . 272
10.137 so_5406533_5911476_1 . . . 272
10.138 so_19712539_19713329_0 . . . 272
10.139 so_19712539_19713329_1 . . . 272
10.140 so_10297405_10300201_0 . . . 272
10.141 so_1030927_1031368_0 . . . 272
10.142 so_22704111_22704617_0 . . . 272
10.143 so_11459473_11460311_0 . . . 272
10.144 so_38874913_38875293_0 . . . 272
10.145 so_30994012_30994298_0 . . . 272

10.146 so_42332147_42332886_0 . . . 272
10.147 so_12143836_12144590_0 . . . 272
10.148 so_23611253_23617539_0 . . . 272
10.149 so_28307323_28307549_0 . . . 272
10.150 so_28307323_28307527_1 . . . 272
10.151 so_47201589_47201947_1 . . . 272
10.152 so_25240985_25242515_0 . . . 272
10.153 so_18398409_18400950_0 . . . 272
10.154 so_19090502_19090722_0 . . . 272
10.155 so_17317083_17317140_0 . . . 272
10.156 so_7098054_7098101_0 . . . 272
10.157 so_17418057_17418213_0 . . . 272
10.158 so_37182808_37182947_0 . . . 272
10.159 so_6035845_6035845_0 . . . 272
10.160 so_33093744_33093821_0 . . . 272
10.161 so_27227189_27227244_0 . . . 272
10.162 so_36635372_36635740_0 . . . 272
10.163 so_10712079_10712116_0 . . . 272
10.164 so_44523336_44523420_1 . . . 272
10.165 so_46485477_46485531_2 . . . 272
10.166 so_43644194_43644563_0 . . . 272
10.167 so_22936579_22936638_1 . . . 272
10.168 so_40249854_40250232_1 . . . 272
10.169 so_20548370_20631809_3 . . . 272
10.170 so_20548370_20631809_4 . . . 272
10.171 so_19736281_19737267_1 . . . 272
10.172 so_19736281_19737267_0 . . . 272
10.173 so_18420645_18420804_0 . . . 272
10.174 so_18420645_18420804_3 . . . 272
10.175 so_18420645_18420804_1 . . . 272
10.176 so_3827374_3827415_1 . . . 272
10.177 so_3827374_3827415_0 . . . 272
10.178 so_43890656_43890796_0 . . . 272
10.179 so_43890656_43890685_2 . . . 272
10.180 so_2669681_2670232_0 . . . 272
10.181 so_31006916_31007045_1 . . . 272
10.182 so_31354825_31354856_0 . . . 272
10.183 so_5309414_5309431_0 . . . 272
10.184 so_1133955_1134198_0 . . . 272
10.185 so_40094929_40094963_2 . . . 272
10.186 so_10488176_10488242_0 . . . 272
10.187 so_30404388_30404389_2 . . . 272
10.188 so_30404388_30404389_0 . . . 272
10.189 so_10607182_10607251_0 . . . 272
10.190 so_4895294_25414336_1 . . . 272
10.191 so_29979527_29979544_1 . . . 272
10.192 so_12918637_12918725_0 . . . 273
10.193 so_29412412_29412622_0 . . . 273
10.194 so_47556287_47556411_2 . . . 273
10.195 so_26543711_26544425_0 . . . 273
10.196 so_44706215_44706352_0 . . . 273

10.197 so_36895012_36898976_0 . . . 273
10.198 so_7538431_7538521_1 . . . 273
10.199 so_26543711_26544425_1 . . . 273
10.200 so_36665029_36665206_1 . . . 274
10.201 so_14184341_14184398_2 . . . 274
10.202 so_28307323_28307527_2 . . . 274
10.203 so_25001147_25001213_2 . . . 274
10.204 so_25001147_25001213_0 . . . 274
10.205 so_25001147_25001213_3 . . . 274
10.206 so_25001147_25001213_1 . . . 274
10.207 so_35958755_35958933_2 . . . 274
10.208 so_23282270_23282403_0 . . . 274
10.209 so_15064247_15064420_3 . . . 274
10.210 so_15064247_15064420_0 . . . 274
10.211 so_15064247_15064420_2 . . . 274
10.212 so_15064247_15064420_1 . . . 274
10.213 so_13212901_13212950_1 . . . 274
10.214 so_25584939_25585130_1 . . . 274
10.215 so_38533734_38533937_0 . . . 274
10.216 so_21963734_21963998_1 . . . 274
10.217 so_21963734_21963998_2 . . . 274
10.218 so_21963734_21963998_3 . . . 274
10.219 so_21963734_21963998_0 . . . 274
10.220 so_4665117_4665155_0 . . . 274
10.221 so_28307323_28307527_3 . . . 274
10.222 so_22502653_22502775_1 . . . 274
10.223 so_1689195_1689267_1 . . . 274
10.224 so_1689195_1689267_0 . . . 274
10.225 so_1689195_1689267_2 . . . 274
10.226 so_1689195_1689267_3 . . . 274
10.227 so_39869216_39869217_0 . . . 274
10.228 so_47172507_47172507_0 . . . 274
10.229 so_18420645_18420804_2 . . . 274
10.230 so_30659377_30659639_0 . . . 274
10.231 so_28132869_30144825_0 . . . 274
10.232 so_26354807_26354955_2 . . . 274
10.233 so_26354807_26354955_0 . . . 274
10.234 so_26354807_26354955_1 . . . 274
10.235 so_26354807_26354955_3 . . . 274
10.236 so_25813435_25813596_0 . . . 274
10.237 so_25813435_25813596_1 . . . 274
10.238 so_17254381_17254443_0 . . . 274
10.239 so_38533734_38533908_1 . . . 274
10.240 so_30252032_30252692_0 . . . 274
10.241 so_1689195_1689267_4 . . . 274
10.242 so_44548242_44549084_3 . . . 274
10.243 so_30126487_30126487_3 . . . 274
10.244 so_28524398_28524697_0 . . . 274
10.245 so_19501838_19502468_0 . . . 274
10.246 so_28132869_30144825_1 . . . 274
10.247 so_37653170_37653674_0 . . . 275

10.248 so_37850882_37851163_0 . . . 275
10.249 so_3599695_3599714_0 . . . 275
10.250 so_4524639_4524670_0 . . . 275
10.251 so_10607182_10607251_1 . . . 275
10.252 so_16377717_16377777_0 . . . 275
10.253 so_23431386_23431606_0 . . . 275
10.254 so_3561648_3562336_2 . . . 275
10.255 so_8931612_8931834_0 . . . 275
10.256 so_10455131_10455390_0 . . . 275
10.257 so_8642124_8642218_0 . . . 275
10.258 so_33526666_33526834_0 . . . 275
10.259 so_10006711_10007510_4 . . . 275
10.260 so_3910083_12101976_1 . . . 275
10.261 so_6466816_26070609_0 . . . 275
10.262 so_46341681_46935513_1 . . . 275
10.263 so_14316494_14316611_0 . . . 275
10.264 so_26811356_26811447_1 . . . 275
10.265 so_9087364_9087411_1 . . . 276
10.266 so_9087364_9087411_0 . . . 276
10.267 so_6615193_6615215_0 . . . 276
10.268 so_24229131_24229538_1 . . . 276
10.269 so_45611464_45613796_5 . . . 276
10.270 so_26243781_26244105_1 . . . 276
10.271 so_26243781_26244105_0 . . . 276
10.272 so_28449304_28449489_0 . . . 276
10.273 so_14365615_14366900_0 . . . 277
10.274 so_3910083_12101976_0 . . . 277
10.275 so_6466816_26070609_1 . . . 277
10.276 Abstract classes . . . 277
10.277 so_23789655_23789770_0 . . . 277
10.278 so_25629061_25630943_0 . . . 277
10.279 so_5722307_5722393_0 . . . 277
10.280 so_10962372_17998615_0 . . . 277
10.281 so_23789655_23789770_1 . . . 277
10.282 so_12592865_17998482_0 . . . 277
10.283 so_42985624_42986059_1 . . . 277
10.284 so_11377237_11378835_0 . . . 277
10.285 so_37017094_37018059_1 . . . 277
10.286 so_23942277_23942317_0 . . . 277
10.287 so_318064_47148505_0 . . . 277
10.288 so_28257485_28257514_0 . . . 277
10.289 so_24229131_24229131_0 . . . 277
10.290 so_8194571_8194625_0 . . . 277
10.291 so_37777606_37778582_1 . . . 277
10.292 so_35422885_35423114_1 . . . 277
10.293 so_13852761_13852825_0 . . . 277
10.294 so_31710196_31710727_0 . . . 277
10.295 so_26243781_26244000_1 . . . 277
10.296 so_16585033_16585719_1 . . . 277
10.297 so_31791103_31791236_2 . . . 277
10.298 so_31791103_31791236_0 . . . 277

10.299 so_31791103_31791236_6 . . . 277
10.300 so_16597389_16597390_1 . . . 277
10.301 so_2951273_2951330_0 . . . 277
10.302 so_4282014_4282044_0 . . . 277
10.303 so_3906384_3906514_0 . . . 277
10.304 so_41536849_41537018_0 . . . 277
10.305 so_29295754_29296413_0 . . . 277
10.306 so_42009795_42010117_0 . . . 277
10.307 so_16597389_16597390_2 . . . 277
10.308 so_16505782_16509511_3 . . . 277
10.309 so_20756473_20756473_0 . . . 277
10.310 so_20548370_20548370_0 . . . 277
10.311 so_2933504_2933525_1 . . . 277
10.312 so_19065836_19065961_0 . . . 278
10.313 so_33526666_33526987_2 . . . 278
10.314 so_16585033_16585719_3 . . . 278
10.315 so_43143733_43145252_1 . . . 278
10.316 so_21968982_21969764_1 . . . 278
10.317 so_2576368_2576395_0 . . . 278
10.318 so_25723041_25723284_1 . . . 278
10.319 so_18456695_18457039_1 . . . 278
10.320 so_18456695_18456764_1 . . . 278
10.321 so_22950500_22950739_0 . . . 278
10.322 so_18456314_18456362_0 . . . 278
10.323 so_24229131_24229202_1 . . . 278
10.324 so_24229131_24229131_2 . . . 278
10.325 so_24229131_24229538_0 . . . 278
10.326 so_42645853_42646077_0 . . . 278
10.327 so_12703142_12703164_1 . . . 278
10.328 so_18456450_18456594_1 . . . 278
10.329 so_14549489_14550864_1 . . . 278
10.330 so_14681349_14682009_0 . . . 278
10.331 so_9212212_9212392_0 . . . 278

11 Member access control **279**
11.1 so_33253245_33253345_1 . . . 279
11.2 so_11809432_11809432_0 . . . 279
11.3 so_29188788_29188959_2 . . . 279
11.4 so_41753597_41753794_2 . . . 279
11.5 so_41753597_41753794_3 . . . 279
11.6 so_41753597_41753794_4 . . . 279
11.7 so_11871944_11872232_0 . . . 279
11.8 so_420727_420799_0 . . . 279
11.9 so_27007831_27008033_4 . . . 279
11.10 so_27007831_27008033_3 . . . 279
11.11 so_31234003_31234270_0 . . . 279
11.12 so_15114505_15114758_0 . . . 279
11.13 so_39144979_39145580_0 . . . 279
11.14 so_39144979_39145580_1 . . . 279
11.15 so_41753597_41753794_6 . . . 279
11.16 so_35885836_35885988_0 . . . 279

11.17 so_5013717_14759603_3 . . . 279
11.18 so_92859_999810_1 . . . 279
11.19 so_10930091_10930149_0 . . . 279
11.20 so_13383351_13384405_0 . . . 279
11.21 so_16206626_16206626_0 . . . 279
11.22 so_39042240_39042473_0 . . . 279
11.23 so_30245091_30245730_3 . . . 279
11.24 so_8366584_8366618_0 . . . 279
11.25 so_8984013_8984264_1 . . . 280
11.26 so_3657494_3657647_0 . . . 280
11.27 so_31234003_31234238_0 . . . 280
11.28 so_33521902_33523142_0 . . . 280
11.29 so_43703229_43839226_0 . . . 280
11.30 so_29951564_29951816_1 . . . 280
11.31 so_41243516_41243516_0 . . . 280
11.32 so_31234003_31234003_0 . . . 280
11.33 so_33521902_33523142_1 . . . 280
11.34 so_24770875_24771100_0 . . . 280
11.35 so_37617717_37618181_0 . . . 280
11.36 Access specifiers . . . 281
11.37 so_14782568_14782790_0 . . . 281
11.38 so_9223441_9223678_0 . . . 281
11.39 so_15342410_15342557_3 . . . 281
11.40 so_47528075_47528556_0 . . . 281
11.41 so_47528075_47528147_1 . . . 281
11.42 so_13927116_13928484_1 . . . 281
11.43 Accessibility of base classes and base class members . . . 282
11.44 so_26994755_26994929_0 . . . 282
11.45 so_29188788_29188959_0 . . . 282
11.46 so_16459646_16459747_0 . . . 282
11.47 so_2421492_2421590_1 . . . 282
11.48 so_13723217_13723325_0 . . . 282
11.49 so_24611629_24611855_0 . . . 282
11.50 so_3900192_3918423_0 . . . 282
11.51 so_4796789_47309154_1 . . . 282
11.52 so_4796789_47309154_0 . . . 282
11.53 so_3811424_3811480_0 . . . 282
11.54 so_16831769_16831867_0 . . . 282
11.55 so_16831769_16832091_0 . . . 282
11.56 so_25166559_25167027_0 . . . 282
11.57 so_4427244_4427244_1 . . . 282
11.58 so_4427244_4427244_0 . . . 282
11.59 so_19609226_19610062_2 . . . 282
11.60 so_92859_999810_0 . . . 282
11.61 so_10930091_10930149_1 . . . 282
11.62 so_10930091_10930239_0 . . . 282
11.63 so_10930091_10930245_1 . . . 282
11.64 so_3047511_3047780_0 . . . 282
11.65 so_46486701_46486701_0 . . . 282
11.66 so_30524479_30526469_3 . . . 283
11.67 so_30524479_30526469_4 . . . 283

11.68 so_30524479_30526469_2 . . . 283
11.69 so_30524479_30526469_1 . . . 283
11.70 so_33087785_33087854_0 . . . 283
11.71 so_31343142_31348802_0 . . . 284
11.72 so_38344470_38345357_0 . . . 284
11.73 so_38344470_38345017_0 . . . 284
11.74 so_38344470_38345357_1 . . . 284
11.75 so_4305597_4305758_2 . . . 284
11.76 so_4305597_4305758_3 . . . 284
11.77 so_4305597_4305758_0 . . . 284
11.78 so_4305597_4305758_4 . . . 284
11.79 so_46646773_46647030_2 . . . 284
11.80 so_31389470_31389692_0 . . . 284
11.81 so_39384399_39384918_0 . . . 284
11.82 so_30524479_30526469_5 . . . 284
11.83 so_24636234_24637204_4 . . . 284
11.84 so_24636234_24637204_3 . . . 284
11.85 so_24636234_24637204_2 . . . 284
11.86 so_24636234_24637204_0 . . . 284
11.87 so_45146025_45146678_0 . . . 284
11.88 so_30626288_30626464_0 . . . 284
11.89 so_10158925_10159182_0 . . . 284
11.90 so_15398558_15421410_0 . . . 284
11.91 so_15398558_15421410_1 . . . 284
11.92 so_3900192_3918423_1 . . . 284
11.93 so_38344470_38345017_1 . . . 284
11.94 Friends . . . 284
11.95 so_23687741_23687847_0 . . . 284
11.96 so_31310429_31310585_5 . . . 284
11.97 so_40770829_40771065_1 . . . 284
11.98 so_3900192_3918423_2 . . . 284
11.99 so_5975421_5975568_0 . . . 285
11.100 so_5975421_5975757_0 . . . 285
11.101 so_23687741_23687847_1 . . . 285
11.102 so_23293561_23294156_0 . . . 285
11.103 so_26162679_26162809_0 . . . 285
11.104 so_20980192_20981414_1 . . . 285
11.105 so_37487609_37487974_0 . . . 285
11.106 so_13985292_13985726_0 . . . 285
11.107 so_30381690_30382232_0 . . . 285
11.108 so_27754396_27754474_0 . . . 285
11.109 so_13985369_13985663_0 . . . 285
11.110 so_36424140_36424958_0 . . . 285
11.111 so_9754084_9754329_1 . . . 285
11.112 so_9754084_9754329_0 . . . 285
11.113 so_26870724_26871006_0 . . . 285
11.114 so_25070030_25070319_0 . . . 286
11.115 so_19845478_19845672_0 . . . 286
11.116 so_13898053_13898517_1 . . . 286
11.117 so_5391927_5392399_3 . . . 286
11.118 so_27663055_27664497_1 . . . 286

11.119 so_22812945_22813383_0 . . . 286
11.120 so_18905090_18905247_0 . . . 286
11.121 so_18310297_18310386_0 . . . 286
11.122 so_5695548_5695585_0 . . . 286
11.123 so_8933699_8934121_0 . . . 286
11.124 so_8207633_8207877_0 . . . 286
11.125 so_33283092_33283499_1 . . . 286
11.126 so_33963794_33964384_0 . . . 286
11.127 so_27376365_27376675_0 . . . 286
11.128 so_5391927_5392399_0 . . . 286
11.129 so_27663055_27664497_2 . . . 286
11.130 so_28241874_28242034_0 . . . 286
11.131 so_25692402_25692496_0 . . . 286
11.132 so_21609006_21609006_0 . . . 286
11.133 so_27477963_27477963_0 . . . 286
11.134 so_5695548_5695585_1 . . . 287
11.135 so_37328444_37328444_0 . . . 287
11.136 so_8933699_8934121_1 . . . 287
11.137 so_8207633_8207877_1 . . . 287
11.138 so_3980627_3980644_0 . . . 287
11.139 so_24600050_24600138_0 . . . 287
11.140 so_42441166_42441827_0 . . . 287
11.141 so_33283092_33283499_2 . . . 287
11.142 so_23540764_23541097_0 . . . 287
11.143 so_23540764_23540764_0 . . . 287
11.144 so_20114685_20114803_0 . . . 287
11.145 so_20114685_20114792_0 . . . 287
11.146 so_27917680_27922878_2 . . . 287
11.147 so_27376365_27376675_1 . . . 287
11.148 so_5391927_5392399_1 . . . 287
11.149 so_27663055_27664497_4 . . . 287
11.150 so_31183484_31183574_0 . . . 287
11.151 so_21609006_21609006_1 . . . 287
11.152 so_4396356_4397350_0 . . . 287
11.153 so_27762167_27762174_0 . . . 287
11.154 so_25144439_25146171_0 . . . 287
11.155 so_5391927_5392399_2 . . . 287
11.156 so_26956176_26956276_0 . . . 287
11.157 so_21134306_21134306_1 . . . 287
11.158 so_7371087_7371096_0 . . . 287
11.159 so_3561648_3562144_0 . . . 287
11.160 so_5968273_5968583_0 . . . 287
11.161 so_23920931_23921069_3 . . . 287
11.162 so_14114956_14115071_0 . . . 287
11.163 so_14114956_14115077_0 . . . 287
11.164 so_6699946_6703063_0 . . . 287
11.165 so_24423481_24424033_2 . . . 287
11.166 so_15424831_15424831_0 . . . 287
11.167 so_8718643_8718857_0 . . . 287
11.168 so_8718643_8718643_0 . . . 287
11.169 Protected member access . . . 288

11.170 so_15702146_15703850_1 . . . 288
11.171 so_19803971_19803971_0 . . . 288
11.172 so_11809432_11809704_0 . . . 288
11.173 so_16913451_16913683_1 . . . 288
11.174 so_16459646_16459747_1 . . . 288
11.175 so_14524278_14524278_0 . . . 288
11.176 so_2594151_2595268_0 . . . 288
11.177 so_29013895_29014122_0 . . . 288
11.178 so_16702224_16702313_0 . . . 288
11.179 so_3057789_3057789_0 . . . 288
11.180 so_38656798_38657465_0 . . . 288
11.181 so_24636234_24636383_0 . . . 288
11.182 so_24636234_24636234_0 . . . 288
11.183 so_24636234_24636383_1 . . . 288
11.184 so_24636234_24636383_2 . . . 288
11.185 so_24636234_24636383_4 . . . 288
11.186 so_24636234_24636383_3 . . . 288
11.187 so_14509291_14509612_0 . . . 288
11.188 so_14400385_14543166_4 . . . 288
11.189 so_14400385_14543166_2 . . . 288
11.190 so_11795349_11795443_1 . . . 288
11.191 so_18272048_18272246_0 . . . 288
11.192 so_41700241_41700345_0 . . . 288
11.193 so_13723217_13723325_1 . . . 288
11.194 so_28622970_28623297_0 . . . 288
11.195 so_29951564_29951816_0 . . . 288
11.196 so_1787643_1787671_0 . . . 288
11.197 so_34588930_34591141_2 . . . 288
11.198 so_34588930_34605638_1 . . . 288
11.199 so_34588930_34591141_1 . . . 288
11.200 so_34588930_34589748_2 . . . 288
11.201 so_38717447_38717508_0 . . . 288
11.202 so_16971897_16971897_0 . . . 288
11.203 Access to virtual functions . . . 289
11.204 so_18185930_18186370_0 . . . 289
11.205 so_34875551_34875841_0 . . . 289
11.206 so_20085756_20085894_0 . . . 289
11.207 so_20905213_20905296_0 . . . 289
11.208 so_41136606_41136930_0 . . . 289
11.209 so_39240085_39241687_1 . . . 289
11.210 so_37879533_37879846_0 . . . 289
11.211 so_43890656_43890685_0 . . . 289
11.212 so_43890656_43890978_0 . . . 289
11.213 so_6038160_6038304_0 . . . 289
11.214 so_34875551_34875841_1 . . . 289
11.215 so_20085756_20085894_1 . . . 289
11.216 so_20905213_20905296_1 . . . 289
11.217 so_41136606_41136930_1 . . . 289
11.218 so_37879533_37879846_3 . . . 289
11.219 so_3610936_3610966_0 . . . 289
11.220 so_6493052_6493117_0 . . . 289

11.221 so_6038160_6038304_1 . . . 289
11.222 so_2141188_2141461_0 . . . 289
11.223 Multiple access . . . 289
11.224 so_8957187_8958112_0 . . . 289
11.225 so_8957187_8958112_1 . . . 290
11.226 Nested classes . . . 290
11.227 so_14801591_14804071_2 . . . 290
11.228 so_11933999_12086873_1 . . . 290
11.229 so_36008654_36008806_0 . . . 290
11.230 so_36008654_36008806_1 . . . 290
11.231 so_5031524_5031773_1 . . . 290
11.232 so_34965014_34965134_0 . . . 290
11.233 so_31310429_31310585_2 . . . 290
11.234 so_21134306_21139534_0 . . . 290
11.235 so_37284762_37285529_1 . . . 290
11.236 so_12063056_12063166_1 . . . 290
11.237 so_23712025_23712633_2 . . . 290
11.238 so_3537969_3537989_1 . . . 290
11.239 so_3537969_3538034_3 . . . 290
11.240 so_3537969_3537998_1 . . . 290
11.241 so_3784652_3784718_2 . . . 290
11.242 so_30006002_32876431_1 . . . 290
11.243 so_30006002_32877675_1 . . . 290
11.244 so_2421492_4528154_1 . . . 290
11.245 so_12868428_12868453_0 . . . 290
11.246 so_8869907_8869944_1 . . . 290
11.247 so_8869907_8870002_0 . . . 290
11.248 so_14607226_14607320_0 . . . 290
11.249 so_14607226_14607402_1 . . . 290
11.250 so_40001188_40001541_1 . . . 290
11.251 so_40001188_40001541_3 . . . 290
11.252 so_42926317_42926466_0 . . . 290
11.253 so_21942236_21942353_0 . . . 290
11.254 so_5013717_5014618_1 . . . 290
11.255 so_5013717_14759603_1 . . . 290
11.256 so_2881305_2881463_1 . . . 290

12 Special member functions **291**
12.1 so_25299995_25300292_3 . . . 291
12.2 so_30634876_30634953_0 . . . 291
12.3 so_44553920_44553920_0 . . . 291
12.4 so_7991666_7991919_2 . . . 291
12.5 so_25821495_25821495_0 . . . 291
12.6 so_5867110_5867173_0 . . . 291
12.7 so_25830995_25831292_4 . . . 291
12.8 so_4172722_4172724_0 . . . 291
12.9 so_26667975_26667997_0 . . . 291
12.10 so_4782757_4782927_0 . . . 291
12.11 so_35461561_35463025_1 . . . 291
12.12 so_3280768_3281119_2 . . . 291
12.13 so_14963163_14963476_0 . . . 291

12.14 so_46402983_46403511_1 . . . 291
12.15 so_33788300_33788346_0 . . . 291
12.16 so_19469565_19480241_1 . . . 291
12.17 so_19469565_19480241_2 . . . 291
12.18 so_30258639_30259298_1 . . . 291
12.19 so_18895784_18895784_0 . . . 291
12.20 so_14963327_14963372_2 . . . 291
12.21 so_6322573_6322591_0 . . . 291
12.22 so_28794443_28794570_0 . . . 291
12.23 so_43703229_43839226_1 . . . 291
12.24 so_7360752_11814437_2 . . . 291
12.25 so_30992462_30992563_4 . . . 291
12.26 so_26986570_26995474_2 . . . 291
12.27 so_30252074_30252333_6 . . . 291
12.28 so_28343407_28343506_1 . . . 291
12.29 so_47559033_47565432_0 . . . 291
12.30 Constructors . . . 291
12.31 so_24373404_24375713_1 . . . 291
12.32 so_40873098_40873098_0 . . . 291
12.33 so_11378515_11378659_0 . . . 292
12.34 so_33842988_33842988_0 . . . 292
12.35 so_33079486_33079574_0 . . . 292
12.36 so_43740520_43740728_3 . . . 292
12.37 so_30603512_30603673_0 . . . 292
12.38 so_28199177_28201770_0 . . . 292
12.39 so_34671852_34672033_1 . . . 292
12.40 so_34329002_34329155_0 . . . 292
12.41 so_24303735_24312655_0 . . . 292
12.42 so_7371020_7371035_1 . . . 292
12.43 so_2823485_2824067_1 . . . 292
12.44 so_41003088_41003364_3 . . . 292
12.45 so_34085147_34085246_0 . . . 292
12.46 so_27668269_27668438_0 . . . 292
12.47 so_21348952_21349017_0 . . . 292
12.48 so_35431651_35433452_0 . . . 292
12.49 so_37255200_37255383_0 . . . 292
12.50 so_8289950_8292768_0 . . . 292
12.51 so_21082581_21082847_0 . . . 292
12.52 so_46353373_46353373_0 . . . 292
12.53 so_8137904_8138772_0 . . . 292
12.54 so_12177632_12177684_0 . . . 292
12.55 so_22284215_22284215_0 . . . 292
12.56 so_2271046_2271055_0 . . . 292
12.57 so_15279508_15279652_0 . . . 292
12.58 so_39583491_39704547_1 . . . 292
12.59 so_12551375_12551434_0 . . . 292
12.60 so_38426461_38426597_0 . . . 292
12.61 so_3258269_3258284_0 . . . 292
12.62 so_1126107_1126995_3 . . . 292
12.63 so_2468367_2468538_3 . . . 292
12.64 so_29150981_29151144_1 . . . 292

12.65 so_24342941_24343032_10 . . . 292
12.66 so_35730462_35730623_2 . . . 292
12.67 so_35730462_35730623_5 . . . 292
12.68 so_35730462_35730623_6 . . . 292
12.69 so_35730462_35730623_3 . . . 292
12.70 so_35730462_35730623_0 . . . 292
12.71 so_35730462_35730623_1 . . . 292
12.72 so_35730462_35730623_4 . . . 292
12.73 so_35730462_35730623_7 . . . 292
12.74 so_6273958_6274011_0 . . . 292
12.75 so_3899223_16233389_0 . . . 292
12.76 so_26994755_26995534_0 . . . 292
12.77 so_3820363_3820388_4 . . . 292
12.78 so_18806954_18807039_0 . . . 292
12.79 so_18806954_18807039_1 . . . 292
12.80 so_14543983_14544093_2 . . . 292
12.81 so_28408911_28409778_8 . . . 292
12.82 so_28408911_28409778_7 . . . 292
12.83 so_28408911_28409778_6 . . . 292
12.84 so_31571202_31571496_1 . . . 292
12.85 so_31571202_31571502_1 . . . 292
12.86 so_22812183_22812183_0 . . . 292
12.87 so_22812183_22816963_0 . . . 292
12.88 so_22812183_22816963_1 . . . 292
12.89 so_22812183_22812183_1 . . . 292
12.90 so_30517546_30517574_7 . . . 292
12.91 so_26221911_26221962_0 . . . 292
12.92 so_21065112_21066022_6 . . . 292
12.93 so_21065112_21066022_3 . . . 292
12.94 so_44950083_44950815_1 . . . 292
12.95 so_13662423_13662485_0 . . . 292
12.96 so_17786033_17786209_0 . . . 292
12.97 so_17786033_17786209_1 . . . 292
12.98 so_20716091_20716155_5 . . . 292
12.99 so_6773723_6774113_0 . . . 292
12.100 so_18278329_18278613_0 . . . 292
12.101 so_7084831_7085249_12 . . . 292
12.102 so_4178175_7189821_17 . . . 292
12.103 so_4178175_7189821_11 . . . 292
12.104 so_26572240_26572651_0 . . . 292
12.105 so_26572240_26572240_1 . . . 292
12.106 so_39745566_39750232_1 . . . 292
12.107 so_41239062_41239457_0 . . . 292
12.108 so_8976722_8976774_0 . . . 292
12.109 so_33809441_33809441_0 . . . 292
12.110 so_5867110_5867215_0 . . . 292
12.111 so_32235294_32235317_0 . . . 292
12.112 so_32235294_32235352_0 . . . 292
12.113 so_16076468_16076878_4 . . . 292
12.114 so_3811424_3811441_0 . . . 292
12.115 so_7626237_7626342_0 . . . 292

12.116 so_12162893_12163119_3 . . . 292
12.117 so_11693512_11694039_1 . . . 292
12.118 so_26418865_26419004_0 . . . 292
12.119 so_13861161_13861611_3 . . . 292
12.120 so_40704924_40705005_0 . . . 292
12.121 so_10788823_10788823_1 . . . 292
12.122 so_30847558_30847558_0 . . . 292
12.123 so_30847558_30847558_1 . . . 292
12.124 so_20806185_20806185_0 . . . 292
12.125 so_11250690_11250719_0 . . . 292
12.126 so_27959496_27959698_1 . . . 292
12.127 so_27959496_27959698_0 . . . 292
12.128 so_27959496_27959703_0 . . . 292
12.129 so_43455035_43455191_0 . . . 292
12.130 so_7054952_19850591_0 . . . 292
12.131 so_21093669_21093669_0 . . . 292
12.132 so_27743264_27743635_0 . . . 292
12.133 so_6524185_6524855_2 . . . 292
12.134 so_23183322_23185451_4 . . . 292
12.135 so_12979529_12979623_4 . . . 292
12.136 so_31482306_31482517_0 . . . 292
12.137 so_42078863_42079254_2 . . . 292
12.138 so_28677593_28685181_0 . . . 292
12.139 so_36098055_36098055_14 . . . 292
12.140 so_36098055_36098055_8 . . . 292
12.141 so_29528102_29528159_0 . . . 292
12.142 so_27200284_27200678_0 . . . 292
12.143 so_27200284_27200678_1 . . . 292
12.144 so_4124271_4124422_2 . . . 292
12.145 so_44094458_44094815_1 . . . 292
12.146 so_44543380_44543581_0 . . . 292
12.147 so_40081920_40082277_1 . . . 292
12.148 so_13922067_13922401_0 . . . 292
12.149 so_13922067_13922235_1 . . . 292
12.150 so_7411515_47368753_1 . . . 292
12.151 so_7411515_7411708_2 . . . 292
12.152 so_33178981_33179101_3 . . . 292
12.153 so_22867654_23087392_7 . . . 292
12.154 so_20828907_20828970_3 . . . 292
12.155 so_20828907_20828970_1 . . . 292
12.156 so_20828907_20828970_2 . . . 292
12.157 so_18123121_18123326_1 . . . 292
12.158 so_42753585_42754529_2 . . . 292
12.159 so_39577922_39577922_3 . . . 292
12.160 so_39577922_39577922_4 . . . 292
12.161 so_42251142_42252328_9 . . . 292
12.162 so_42251142_42252328_8 . . . 292
12.163 so_42251142_42252328_7 . . . 292
12.164 so_42251142_42252328_10 . . . 292
12.165 so_42251142_42252328_6 . . . 292
12.166 so_28133607_28133664_0 . . . 292

12.167 so_2953611_2953925_0 . . . 292
12.168 so_36354041_36354041_1 . . . 292
12.169 so_36354041_36354149_2 . . . 292
12.170 so_28498880_28498931_0 . . . 292
12.171 so_31766613_31766689_0 . . . 292
12.172 so_18081114_18081330_0 . . . 292
12.173 so_18081114_18081262_0 . . . 292
12.174 so_14941723_14941807_5 . . . 292
12.175 so_24237329_24237615_6 . . . 292
12.176 so_33376124_33376124_3 . . . 292
12.177 so_33345602_33345633_0 . . . 292
12.178 so_3580789_3580876_3 . . . 292
12.179 so_46402983_46403511_2 . . . 292
12.180 so_39057240_39057301_1 . . . 292
12.181 so_4181059_4181132_3 . . . 292
12.182 so_39904101_39904258_3 . . . 292
12.183 so_14566341_14566795_1 . . . 292
12.184 so_14566341_14566807_1 . . . 292
12.185 so_14566341_14566784_0 . . . 292
12.186 so_14566341_14566807_0 . . . 292
12.187 so_14566341_14566795_0 . . . 292
12.188 so_14566341_14566341_1 . . . 292
12.189 so_23107568_23107664_0 . . . 292
12.190 so_5440154_5440235_0 . . . 292
12.191 so_4275861_4417899_1 . . . 292
12.192 so_4275861_4417899_0 . . . 292
12.193 so_30252074_30252333_4 . . . 292
12.194 so_10402241_10402308_7 . . . 292
12.195 so_19347729_19347782_0 . . . 292
12.196 so_33124542_33125217_1 . . . 292
12.197 so_30152687_30152747_0 . . . 292
12.198 so_30152687_30152747_1 . . . 292
12.199 so_18313509_18313750_0 . . . 292
12.200 so_24167652_24167902_7 . . . 292
12.201 so_24167652_24167902_13 . . . 292
12.202 so_24167652_24167902_12 . . . 292
12.203 so_24167652_24167902_6 . . . 292
12.204 so_24082334_24082460_0 . . . 292
12.205 so_28332078_28332609_0 . . . 292
12.206 so_19606492_19607079_0 . . . 292
12.207 so_24043897_24043897_0 . . . 292
12.208 so_23304025_23304090_7 . . . 292
12.209 so_23304025_23304090_6 . . . 292
12.210 so_10638527_10638645_0 . . . 292
12.211 so_26942887_26942887_2 . . . 292
12.212 so_26942887_26949603_2 . . . 292
12.213 so_40106941_40107282_1 . . . 292
12.214 so_25324582_25325041_9 . . . 292
12.215 so_40001188_40001541_0 . . . 292
12.216 so_6310638_6310703_7 . . . 292
12.217 so_19843816_19868173_5 . . . 292

12.218 so_33978185_33978267_0 . . . 292
12.219 so_27349679_27349679_6 . . . 292
12.220 so_7469468_7469610_0 . . . 292
12.221 so_7469468_7469468_0 . . . 292
12.222 so_32489702_32490017_1 . . . 292
12.223 so_26923320_26923320_2 . . . 292
12.224 so_26923320_26923549_5 . . . 292
12.225 so_28636352_28636515_1 . . . 292
12.226 so_28636352_28636515_0 . . . 292
12.227 so_2624628_2624947_3 . . . 292
12.228 so_16542291_16542385_9 . . . 292
12.229 so_92859_2661721_3 . . . 292
12.230 so_1888293_1888452_0 . . . 292
12.231 so_1888293_1888337_0 . . . 292
12.232 so_33254582_33255086_2 . . . 292
12.233 so_1751378_1751546_3 . . . 292
12.234 so_27646565_27647126_0 . . . 292
12.235 so_27646565_27646586_0 . . . 292
12.236 so_23920931_23921069_0 . . . 292
12.237 so_23007358_23007559_4 . . . 292
12.238 so_23007358_23007559_2 . . . 292
12.239 so_23007358_23007559_5 . . . 292
12.240 so_23007358_23007559_1 . . . 292
12.241 so_23007358_23007559_3 . . . 292
12.242 so_39139087_39139188_2 . . . 292
12.243 so_4836596_4836611_0 . . . 292
12.244 so_23005241_23401282_15 . . . 292
12.245 so_23005241_23401282_2 . . . 292
12.246 so_23005241_23401282_14 . . . 292
12.247 so_23005241_23401282_18 . . . 292
12.248 so_23005241_23401282_17 . . . 292
12.249 so_23005241_23401282_5 . . . 292
12.250 so_23005241_23401282_6 . . . 292
12.251 so_23005241_23401282_1 . . . 292
12.252 so_23005241_23401282_0 . . . 292
12.253 so_23005241_23401282_3 . . . 292
12.254 so_23005241_23401282_9 . . . 292
12.255 so_23005241_23401282_8 . . . 292
12.256 so_31398388_31398779_5 . . . 292
12.257 so_31398388_31398779_6 . . . 292
12.258 so_26754024_26754247_2 . . . 292
12.259 so_29718472_29718523_0 . . . 292
12.260 so_26947704_26949366_4 . . . 292
12.261 so_23698203_23698451_0 . . . 293
12.262 so_35256315_35256813_0 . . . 293
12.263 so_26182734_26182783_0 . . . 293
12.264 so_28436148_28583523_1 . . . 293
12.265 so_33813929_33813930_3 . . . 293
12.266 so_16612191_16612281_0 . . . 293
12.267 so_5867110_5867215_2 . . . 293
12.268 so_38762157_38764809_1 . . . 293

12.269 so_43455035_43455191_1 . . . 293
12.270 so_24836251_24836257_0 . . . 293
12.271 so_14267399_14267801_0 . . . 293
12.272 so_43782361_43782808_1 . . . 293
12.273 so_13922067_13922235_0 . . . 293
12.274 so_20828907_20828970_0 . . . 293
12.275 so_3280768_3281119_1 . . . 293
12.276 so_24333458_24333884_3 . . . 293
12.277 so_29307521_29307679_0 . . . 293
12.278 so_5049891_5049921_0 . . . 293
12.279 so_17430377_17433978_0 . . . 293
12.280 so_10917875_10917935_0 . . . 293
12.281 so_20810378_20810439_0 . . . 293
12.282 so_19347729_19347782_1 . . . 293
12.283 so_33441087_33441250_0 . . . 293
12.284 so_24082334_24082460_1 . . . 293
12.285 so_43962934_43965247_0 . . . 293
12.286 so_17738439_17738467_0 . . . 293
12.287 so_18901315_18902462_4 . . . 293
12.288 so_14659752_14659931_4 . . . 293
12.289 so_8218397_8218460_2 . . . 293
12.290 so_27646565_27647126_2 . . . 293
12.291 so_3900192_3918423_3 . . . 293
12.292 so_3900192_3918423_5 . . . 293
12.293 so_33260200_33284388_0 . . . 293
12.294 so_42579031_42590018_0 . . . 293
12.295 so_6773723_6774113_1 . . . 293
12.296 so_21352507_21352605_2 . . . 293
12.297 so_43071676_43071676_2 . . . 293
12.298 so_18123121_18123326_3 . . . 293
12.299 so_31927128_31927151_0 . . . 293
12.300 so_17238906_17239100_0 . . . 293
12.301 so_3870649_3870711_0 . . . 293
12.302 so_1758142_1765193_0 . . . 293
12.303 so_8738929_8738929_0 . . . 293
12.304 so_20829244_20829244_0 . . . 293
12.305 so_30258639_30259298_4 . . . 293
12.306 Temporary objects . . . 294
12.307 so_8586108_8586108_0 . . . 294
12.308 so_45698374_45772026_0 . . . 294
12.309 so_42043414_42043414_2 . . . 294
12.310 so_8760813_8760854_0 . . . 294
12.311 so_25663642_25663854_0 . . . 294
12.312 so_42441791_42442265_1 . . . 294
12.313 so_8923018_8923762_3 . . . 294
12.314 so_919701_919968_0 . . . 294
12.315 so_33433337_33433337_0 . . . 294
12.316 so_3249147_3249392_5 . . . 294
12.317 so_17508703_17637679_0 . . . 294
12.318 so_1517703_1518278_0 . . . 294
12.319 so_2222110_2222255_1 . . . 294

12.320 so_353632_353649_1 . . . 294
12.321 so_8804612_8808865_2 . . . 294
12.322 so_35041577_35041790_1 . . . 294
12.323 so_42055583_42097383_2 . . . 294
12.324 so_24238786_24239020_2 . . . 294
12.325 so_45284117_45284117_0 . . . 294
12.326 so_21554577_21554578_6 . . . 294
12.327 so_22770318_22773745_1 . . . 294
12.328 so_23830767_23831467_1 . . . 294
12.329 so_1032243_1032516_0 . . . 294
12.330 so_45415955_45415955_0 . . . 294
12.331 so_45415955_45415955_1 . . . 294
12.332 so_12686099_12686210_0 . . . 294
12.333 so_1032243_1032516_1 . . . 294
12.334 so_28280169_28280284_0 . . . 295
12.335 so_34592923_34592975_0 . . . 295
12.336 so_16533295_16533353_0 . . . 295
12.337 so_3097806_3097889_2 . . . 295
12.338 so_16291127_16292092_0 . . . 295
12.339 so_15410935_15411077_1 . . . 295
12.340 so_7118323_7118360_1 . . . 295
12.341 so_4863324_4863373_0 . . . 295
12.342 so_7401521_7401782_0 . . . 295
12.343 so_1714874_1714974_0 . . . 295
12.344 so_36328193_36328312_0 . . . 295
12.345 so_8923018_8923762_4 . . . 295
12.346 so_26304239_26304339_0 . . . 295
12.347 so_18146267_18146466_1 . . . 295
12.348 so_5760866_5761101_0 . . . 295
12.349 so_5760866_5761142_0 . . . 295
12.350 so_27195825_27195973_0 . . . 295
12.351 so_34673460_34673980_2 . . . 295
12.352 so_42580307_42581330_0 . . . 295
12.353 so_14169752_14169904_0 . . . 295
12.354 so_15015758_15015997_0 . . . 295
12.355 so_13643649_13644612_0 . . . 295
12.356 so_23557419_23557641_0 . . . 295
12.357 so_26324347_26324347_0 . . . 295
12.358 so_31659123_31659788_0 . . . 295
12.359 so_40735379_40735526_0 . . . 295
12.360 so_22146647_22146739_1 . . . 295
12.361 so_22146647_22146708_0 . . . 295
12.362 so_35669402_35669600_1 . . . 295
12.363 so_35669402_35670062_0 . . . 295
12.364 so_35669402_35669525_0 . . . 295
12.365 so_14251010_14251032_0 . . . 295
12.366 so_36066000_36066446_0 . . . 295
12.367 so_18401345_18401479_1 . . . 295
12.368 so_46183397_46183457_0 . . . 295
12.369 so_2087600_2088683_0 . . . 295
12.370 so_15580903_15580919_0 . . . 295

12.371 so_33166843_33166917_0 . . . 295
12.372 so_5733180_5733237_0 . . . 295
12.373 so_38345484_38345852_1 . . . 295
12.374 so_26793315_26793677_0 . . . 295
12.375 so_21554577_21554578_5 . . . 295
12.376 so_30329758_30330052_0 . . . 295
12.377 so_84427_1152218_1 . . . 295
12.378 so_3870649_3870711_2 . . . 295
12.379 so_3041249_3041264_0 . . . 295
12.380 so_2506793_2507225_0 . . . 295
12.381 so_13474486_13474756_0 . . . 295
12.382 so_27131378_27131690_0 . . . 295
12.383 so_30519148_30519155_0 . . . 295
12.384 so_21205489_21205834_1 . . . 295
12.385 so_38954300_38954300_2 . . . 295
12.386 so_42868744_42868831_3 . . . 295
12.387 so_41928634_41928634_1 . . . 295
12.388 so_19306984_19306984_0 . . . 295
12.389 so_12551884_12551944_0 . . . 295
12.390 so_3097806_3097889_3 . . . 295
12.391 so_26212338_26212515_0 . . . 295
12.392 so_21273570_21274897_0 . . . 295
12.393 so_16291127_16292092_2 . . . 295
12.394 so_7118323_7118360_2 . . . 295
12.395 so_40167231_40168133_1 . . . 295
12.396 so_1714874_1714974_1 . . . 295
12.397 so_26357392_26358037_0 . . . 295
12.398 so_14612933_14612972_0 . . . 295
12.399 so_31796511_31796615_0 . . . 295
12.400 so_28475630_28487646_0 . . . 295
12.401 so_38226443_38226931_0 . . . 295
12.402 so_38226443_38226983_0 . . . 295
12.403 so_2222110_2222255_2 . . . 295
12.404 so_42580307_42581330_2 . . . 295
12.405 so_30489726_30489798_0 . . . 295
12.406 so_30448182_30448214_0 . . . 295
12.407 so_29162620_29168643_4 . . . 295
12.408 so_29162620_29168643_7 . . . 295
12.409 so_35947296_35947465_0 . . . 295
12.410 so_40735379_40735526_1 . . . 295
12.411 so_12554619_12554621_0 . . . 295
12.412 so_32813694_32813750_0 . . . 295
12.413 so_42055583_42097383_0 . . . 295
12.414 so_37718495_37718726_0 . . . 295
12.415 so_3199067_3300627_0 . . . 295
12.416 so_39025342_39025342_0 . . . 295
12.417 so_43354360_43354670_0 . . . 295
12.418 so_33846664_33848134_0 . . . 295
12.419 so_701699_701775_1 . . . 295
12.420 so_21045518_21046043_4 . . . 295
12.421 so_6315670_6315670_0 . . . 295

12.422 so_26793315_26793677_1 . . . 295
12.423 so_36676419_36676909_0 . . . 295
12.424 so_14989183_14989472_0 . . . 295
12.425 so_2506793_2507225_1 . . . 295
12.426 so_8450898_8450987_0 . . . 295
12.427 so_27131378_27131690_1 . . . 295
12.428 so_30519148_30519155_1 . . . 295
12.429 so_15267676_15267760_1 . . . 295
12.430 so_17236007_24210020_0 . . . 295
12.431 so_32665221_32665395_0 . . . 295
12.432 so_32665221_32665221_0 . . . 295
12.433 so_21205489_21205834_3 . . . 295
12.434 so_21205489_21205834_2 . . . 295
12.435 so_2604206_2604269_0 . . . 295
12.436 so_33301552_33302195_3 . . . 295
12.437 so_42868744_42868831_4 . . . 295
12.438 so_14309633_14309880_0 . . . 295
12.439 so_41928634_41928634_3 . . . 295
12.440 so_41928634_41928634_4 . . . 295
12.441 so_41928634_41928634_2 . . . 295
12.442 so_41928634_41928634_6 . . . 295
12.443 so_41928634_41928634_5 . . . 295
12.444 so_35313292_35313292_0 . . . 295
12.445 so_35313292_35335037_0 . . . 295
12.446 so_35313292_35335037_1 . . . 295
12.447 so_44089539_44089539_1 . . . 295
12.448 so_19306984_19306984_1 . . . 295
12.449 so_30618783_30619210_0 . . . 295
12.450 so_30618783_30619210_2 . . . 295
12.451 so_32223429_32223783_0 . . . 295
12.452 so_30488692_30488841_0 . . . 295
12.453 so_26283076_26283564_0 . . . 295
12.454 so_23874451_47288940_2 . . . 295
12.455 so_23874451_47288940_1 . . . 295
12.456 so_12551884_12551944_1 . . . 295
12.457 so_12551884_12551944_2 . . . 295
12.458 so_3097806_3097889_4 . . . 295
12.459 so_3097806_3097889_5 . . . 295
12.460 so_26212338_26212515_1 . . . 295
12.461 so_21273570_21274897_4 . . . 295
12.462 so_21273570_21274897_8 . . . 295
12.463 so_21273570_21274897_6 . . . 295
12.464 so_21273570_21274897_3 . . . 295
12.465 so_21273570_21274897_5 . . . 295
12.466 so_21273570_21274897_1 . . . 295
12.467 so_21273570_21274897_7 . . . 295
12.468 so_21273570_21274897_2 . . . 295
12.469 so_16291127_16292092_3 . . . 295
12.470 so_7118323_7118360_3 . . . 295
12.471 so_14735630_14738884_0 . . . 295
12.472 so_14735630_14735632_0 . . . 295

12.473 so_40167231_40168133_2 . . . 295
12.474 so_37577080_37585451_0 . . . 295
12.475 so_23970565_23970566_4 . . . 295
12.476 so_23970565_23970566_6 . . . 295
12.477 so_23970565_23970566_5 . . . 295
12.478 so_23970565_23970566_9 . . . 295
12.479 so_23970565_23970566_2 . . . 295
12.480 so_23970565_23970566_8 . . . 295
12.481 so_34081317_34082287_7 . . . 295
12.482 so_4863324_4863373_1 . . . 295
12.483 so_4863324_4863373_2 . . . 295
12.484 so_28912679_28912679_0 . . . 295
12.485 so_30559817_30560064_1 . . . 295
12.486 so_1714874_1714974_2 . . . 295
12.487 so_19790804_19791281_3 . . . 295
12.488 so_19790804_19791281_7 . . . 295
12.489 so_26357392_26358037_1 . . . 295
12.490 so_42441791_42441832_0 . . . 295
12.491 so_42441791_42442265_2 . . . 295
12.492 so_28176415_28176643_5 . . . 295
12.493 so_37521150_37521808_0 . . . 295
12.494 so_14612933_14612972_2 . . . 295
12.495 so_24760041_24760066_2 . . . 295
12.496 so_2615162_2615199_0 . . . 295
12.497 so_26484734_26484734_0 . . . 295
12.498 so_26484734_26486274_0 . . . 295
12.499 so_34091623_34369191_0 . . . 295
12.500 so_4141278_4142094_0 . . . 295
12.501 so_10190677_10190677_0 . . . 295
12.502 so_10190677_10194011_2 . . . 295
12.503 so_10190677_10194011_0 . . . 295
12.504 so_33951129_33952280_1 . . . 295
12.505 so_33951129_33952333_0 . . . 295
12.506 so_33951129_33952280_0 . . . 295
12.507 so_28717682_28718438_5 . . . 295
12.508 so_30461093_30461305_1 . . . 295
12.509 so_30461093_30461305_0 . . . 295
12.510 so_25333462_25334005_0 . . . 295
12.511 so_32738281_32738442_2 . . . 295
12.512 so_32738281_32738442_0 . . . 295
12.513 so_31796511_31796615_1 . . . 295
12.514 so_31404920_31405006_2 . . . 295
12.515 so_31404920_31405006_1 . . . 295
12.516 so_31404920_31405006_0 . . . 295
12.517 so_28475630_28487646_1 . . . 295
12.518 so_39446854_39446854_0 . . . 295
12.519 so_38226443_38227096_0 . . . 295
12.520 so_38226443_38226948_2 . . . 295
12.521 so_38226443_38226948_3 . . . 295
12.522 so_38226443_38226931_1 . . . 295
12.523 so_38226443_38226983_1 . . . 295

12.524 so_38226443_38226948_1 . . . 295
12.525 so_19808749_19808808_0 . . . 295
12.526 so_25595742_25598405_3 . . . 295
12.527 so_25595742_25598405_2 . . . 295
12.528 so_46506402_46506881_1 . . . 295
12.529 so_7494318_7494424_0 . . . 295
12.530 so_38334726_38334827_1 . . . 295
12.531 so_27019340_27020187_0 . . . 295
12.532 so_30489726_30489798_1 . . . 295
12.533 so_4985800_4985978_0 . . . 295
12.534 so_30448182_30448214_1 . . . 295
12.535 so_29162620_29168643_8 . . . 295
12.536 so_29162620_29168643_5 . . . 295
12.537 so_29162620_29168643_2 . . . 295
12.538 so_21481481_21481630_0 . . . 295
12.539 so_21481481_21481630_1 . . . 295
12.540 so_21481481_21481614_0 . . . 295
12.541 so_22919725_22920219_1 . . . 295
12.542 so_35947296_35947513_2 . . . 295
12.543 so_35947296_35947513_3 . . . 295
12.544 so_35947296_35947513_1 . . . 295
12.545 so_35947296_35947513_0 . . . 295
12.546 so_35947296_35947465_1 . . . 295
12.547 so_35947296_35947513_4 . . . 295
12.548 so_31406495_31406624_0 . . . 295
12.549 so_40735379_40735526_4 . . . 295
12.550 so_40735379_40735526_5 . . . 295
12.551 so_40735379_40735526_2 . . . 295
12.552 so_40735379_40735526_3 . . . 295
12.553 so_37518434_37518676_1 . . . 295
12.554 so_12554619_12554621_1 . . . 295
12.555 so_25113603_25114728_0 . . . 295
12.556 so_42763741_42764064_1 . . . 295
12.557 so_24346841_24346890_1 . . . 295
12.558 so_28291101_28291138_0 . . . 295
12.559 so_2784262_2784304_0 . . . 295
12.560 so_28318799_28323698_11 . . . 295
12.561 so_32813694_32813750_1 . . . 295
12.562 so_26802048_26802048_0 . . . 295
12.563 so_14490877_14490951_0 . . . 295
12.564 so_42055583_42097383_1 . . . 295
12.565 so_16132252_16132283_0 . . . 295
12.566 so_23973439_23973577_4 . . . 295
12.567 so_23973439_23973577_7 . . . 295
12.568 so_23973439_23973577_8 . . . 295
12.569 so_23973439_23973577_6 . . . 295
12.570 so_23973439_23973577_1 . . . 295
12.571 so_23973439_23973577_3 . . . 295
12.572 so_23973439_23973577_2 . . . 295
12.573 so_23973439_23973577_5 . . . 295
12.574 so_21920311_21920311_2 . . . 295

12.575 so_21920311_21920311_3 . . . 295
12.576 so_21920311_21920311_4 . . . 295
12.577 so_21920311_21920311_1 . . . 295
12.578 so_21920311_21920311_0 . . . 295
12.579 so_27423364_27423503_0 . . . 295
12.580 so_23892018_23965233_10 . . . 295
12.581 so_23892018_23965233_13 . . . 295
12.582 so_23892018_23965233_14 . . . 295
12.583 so_23892018_23965233_8 . . . 295
12.584 so_23892018_23965233_12 . . . 295
12.585 so_23892018_23965233_11 . . . 295
12.586 so_23892018_23965233_9 . . . 295
12.587 so_23892018_23965233_7 . . . 295
12.588 so_7386628_7386628_1 . . . 295
12.589 so_7386628_7386628_2 . . . 295
12.590 so_7386628_7386628_0 . . . 295
12.591 so_3199067_3300627_1 . . . 295
12.592 so_43354360_43354670_2 . . . 295
12.593 so_43354360_43354683_0 . . . 295
12.594 so_43354360_43354670_4 . . . 295
12.595 so_43354360_43354670_1 . . . 295
12.596 so_43354360_43354670_3 . . . 295
12.597 so_33846664_33846664_0 . . . 295
12.598 so_33846664_33848134_1 . . . 295
12.599 so_43349808_43350607_1 . . . 295
12.600 so_43349808_43350607_3 . . . 295
12.601 so_43349808_43350607_4 . . . 295
12.602 so_43349808_43349808_0 . . . 295
12.603 so_43349808_43350607_5 . . . 295
12.604 so_43349808_43350607_2 . . . 295
12.605 so_701699_701775_4 . . . 295
12.606 so_701699_701775_3 . . . 295
12.607 so_701699_701775_2 . . . 295
12.608 so_21045518_21046043_5 . . . 295
12.609 so_6315670_6315670_1 . . . 295
12.610 so_20546981_20547296_0 . . . 295
12.611 so_38345484_38345852_2 . . . 295
12.612 so_15094416_15094480_0 . . . 295
12.613 so_30600231_30600273_0 . . . 295
12.614 so_39449946_39451728_0 . . . 295
12.615 so_15513734_15514013_0 . . . 295
12.616 so_17362673_17362673_0 . . . 295
12.617 so_28875810_28875810_0 . . . 295
12.618 so_26793315_26793677_2 . . . 295
12.619 so_26793315_26793677_4 . . . 295
12.620 so_31206988_31206988_6 . . . 295
12.621 so_27124159_27124621_3 . . . 295
12.622 so_37549377_37549377_0 . . . 295
12.623 so_9657708_9657748_2 . . . 295
12.624 so_9657708_9657797_0 . . . 295
12.625 so_27237423_27237787_2 . . . 295

12.626 so_5689463_5689588_0 . . . 295
12.627 so_16714595_16714670_0 . . . 295
12.628 so_84427_1152218_3 . . . 295
12.629 so_35315826_35316261_0 . . . 295
12.630 so_6936720_6976563_0 . . . 295
12.631 so_6936720_6937829_1 . . . 295
12.632 so_6936720_6937056_0 . . . 295
12.633 so_6936720_17354675_1 . . . 295
12.634 so_6936720_17354675_0 . . . 295
12.635 so_36676419_36676909_1 . . . 295
12.636 so_17353560_17354296_0 . . . 295
12.637 so_17353560_17353560_0 . . . 295
12.638 so_33790065_33790696_2 . . . 295
12.639 so_3870649_3870711_3 . . . 295
12.640 so_38731039_38731039_5 . . . 295
12.641 so_38731039_38731039_3 . . . 295
12.642 so_38731039_38731039_1 . . . 295
12.643 so_45293841_45294129_1 . . . 295
12.644 so_45293841_45294129_0 . . . 295
12.645 so_11747652_11747735_1 . . . 295
12.646 so_23685460_23685460_0 . . . 295
12.647 so_14989183_14989472_3 . . . 295
12.648 so_14989183_14989472_1 . . . 295
12.649 so_14989183_14989472_2 . . . 295
12.650 so_14989183_14989472_4 . . . 295
12.651 so_14989183_14989472_5 . . . 295
12.652 so_2506793_2506811_0 . . . 295
12.653 so_8450898_8450995_0 . . . 295
12.654 so_8450898_8450987_1 . . . 295
12.655 so_15286450_17635374_2 . . . 295
12.656 so_14968569_14968668_1 . . . 295
12.657 so_14419616_14419616_1 . . . 295
12.658 so_14419616_14419616_0 . . . 295
12.659 Conversions . . . 296
12.660 so_17339895_17348767_0 . . . 296
12.661 so_26244887_26245704_10 . . . 296
12.662 so_26244887_26245704_9 . . . 296
12.663 so_34490247_34490537_3 . . . 296
12.664 so_35790664_35790867_4 . . . 296
12.665 so_28265991_28266053_0 . . . 296
12.666 so_12847272_12847286_0 . . . 296
12.667 so_10320282_10320385_0 . . . 296
12.668 so_26824706_26824747_0 . . . 296
12.669 so_867462_867804_0 . . . 296
12.670 so_26341835_26342042_0 . . . 296
12.671 so_11288981_11289102_0 . . . 296
12.672 so_30315254_30315317_0 . . . 296
12.673 so_34147715_34147799_2 . . . 296
12.674 so_29898613_29899468_1 . . . 296
12.675 so_31772588_31773372_0 . . . 296
12.676 so_22444835_22444835_0 . . . 296

12.677 so_35790664_35790867_5 . . . 296
12.678 so_22444415_22444656_0 . . . 296
12.679 so_28265991_28266053_1 . . . 296
12.680 so_12847272_12847286_1 . . . 296
12.681 so_867462_867804_1 . . . 296
12.682 so_13434672_13543853_1 . . . 296
12.683 so_29898613_29899468_2 . . . 296
12.684 so_3895647_3895647_2 . . . 296
12.685 so_9661078_9661078_0 . . . 296
12.686 so_28265991_28266053_2 . . . 296
12.687 so_12847272_12847286_2 . . . 296
12.688 so_867462_867804_2 . . . 296
12.689 so_17339895_17348767_1 . . . 296
12.690 so_41935689_41935790_0 . . . 296
12.691 so_38461522_38461987_0 . . . 296
12.692 so_45917634_45917775_0 . . . 296
12.693 so_35790664_35790867_7 . . . 296
12.694 so_35790664_35790854_0 . . . 296
12.695 so_28265991_28266053_3 . . . 296
12.696 so_18895737_18895841_1 . . . 296
12.697 so_34834986_34838255_3 . . . 296
12.698 so_12847272_12847286_3 . . . 296
12.699 so_867462_867462_0 . . . 296
12.700 so_867462_867804_3 . . . 296
12.701 so_26341835_26342042_2 . . . 296
12.702 so_21017692_21018308_1 . . . 296
12.703 so_30315254_30315317_1 . . . 296
12.704 so_35374810_35374810_0 . . . 296
12.705 so_12677711_12732730_0 . . . 296
12.706 so_13434672_13543853_2 . . . 297
12.707 so_25241516_25242022_0 . . . 297
12.708 so_25241516_25242107_4 . . . 297
12.709 so_3814865_3860424_0 . . . 297
12.710 so_24087187_24087291_0 . . . 297
12.711 so_42472795_42472844_0 . . . 297
12.712 so_31514057_31514112_0 . . . 297
12.713 so_31514057_31514112_1 . . . 297
12.714 so_15077466_15077788_1 . . . 297
12.715 so_15077466_15077788_0 . . . 297
12.716 so_121162_31351956_1 . . . 297
12.717 so_29381171_29381331_3 . . . 297
12.718 so_7006202_7006249_7 . . . 297
12.719 so_26612554_26612762_3 . . . 297
12.720 so_36722011_36722513_0 . . . 297
12.721 so_29834308_29834426_3 . . . 297
12.722 so_35336990_35337463_2 . . . 297
12.723 so_25795442_25795685_2 . . . 297
12.724 so_8464400_8585258_4 . . . 297
12.725 so_12185094_12185546_0 . . . 297
12.726 so_24087187_24087291_1 . . . 297
12.727 so_24381006_24381472_0 . . . 297

12.728 so_6775336_6775552_1 . . . 297
12.729 so_15077466_15077788_4 . . . 297
12.730 so_4153527_4154213_0 . . . 297
12.731 so_121162_31351956_2 . . . 297
12.732 so_7006202_7006249_9 . . . 297
12.733 so_36722011_36722513_1 . . . 297
12.734 so_2836939_2836939_1 . . . 297
12.735 so_32427433_32428111_0 . . . 297
12.736 so_28423522_28423542_0 . . . 297
12.737 so_4801136_4801211_0 . . . 297
12.738 so_28010392_28010428_0 . . . 297
12.739 so_12428308_12428308_1 . . . 298
12.740 so_12428308_12428308_0 . . . 298
12.741 so_35256315_35256813_1 . . . 298
12.742 so_18405501_18405796_0 . . . 298
12.743 so_32129263_32130042_0 . . . 298
12.744 so_28131172_28131248_0 . . . 298
12.745 so_25263128_25263258_0 . . . 298
12.746 so_3814865_3860424_1 . . . 298
12.747 so_3814865_3860424_3 . . . 298
12.748 so_35069224_35069579_1 . . . 298
12.749 so_35069224_35069579_0 . . . 298
12.750 so_9913834_9913985_0 . . . 298
12.751 so_4031228_4031487_0 . . . 298
12.752 so_24836062_24836148_3 . . . 298
12.753 so_31673850_31674621_1 . . . 298
12.754 so_25303662_25303864_1 . . . 298
12.755 so_28657392_28657392_0 . . . 298
12.756 so_28657392_28657392_1 . . . 298
12.757 so_11581174_11582064_1 . . . 298
12.758 so_11581174_11582064_0 . . . 298
12.759 so_28656779_28656779_0 . . . 298
12.760 so_31839730_31839803_0 . . . 298
12.761 so_8350678_8351375_0 . . . 298
12.762 so_9622089_9636595_0 . . . 298
12.763 so_44864106_44864483_0 . . . 298
12.764 so_36817129_36817129_0 . . . 298
12.765 so_36817129_36817129_2 . . . 298
12.766 so_36817129_36817129_5 . . . 298
12.767 so_36817129_36817129_3 . . . 298
12.768 so_36585964_36586174_0 . . . 298
12.769 so_42373862_42373862_1 . . . 298
12.770 so_14117045_14117166_0 . . . 298
12.771 so_7329617_7338579_1 . . . 298
12.772 so_7329617_7338579_0 . . . 298
12.773 so_11794318_11795127_1 . . . 298
12.774 so_23272181_23272181_7 . . . 298
12.775 so_23272181_23272181_0 . . . 298
12.776 so_23272181_23273284_7 . . . 298
12.777 so_23272181_23273284_18 . . . 298
12.778 so_25123709_25123809_0 . . . 298

12.779 so_25123709_25123809_1 . . . 298
12.780 so_26994485_26994566_0 . . . 298
12.781 so_3926929_6614779_2 . . . 298
12.782 so_36132032_36132428_0 . . . 298
12.783 so_19356232_19356589_0 . . . 298
12.784 so_8336274_8336274_0 . . . 298
12.785 so_24161253_24161758_0 . . . 298
12.786 so_24161253_24161758_1 . . . 298
12.787 so_25598299_25598419_7 . . . 298
12.788 so_25598299_25598419_4 . . . 298
12.789 so_22566388_22566777_1 . . . 298
12.790 so_41954449_41954449_1 . . . 298
12.791 so_12372442_12372442_1 . . . 299
12.792 so_23364606_23365816_0 . . . 299
12.793 so_14720913_14721483_0 . . . 299
12.794 so_6207936_6207966_0 . . . 299
12.795 so_20498142_20687366_1 . . . 299
12.796 so_121162_31351956_4 . . . 299
12.797 so_42613866_42614016_2 . . . 299
12.798 so_22603979_22604146_2 . . . 299
12.799 so_22603979_22604146_1 . . . 299
12.800 so_27573928_27574235_0 . . . 299
12.801 so_45322138_45322372_2 . . . 299
12.802 so_44242395_44242881_0 . . . 299
12.803 so_24836062_24836148_1 . . . 299
12.804 so_26029277_26029456_7 . . . 299
12.805 so_10166328_10166328_0 . . . 299
12.806 so_26566090_26566275_2 . . . 299
12.807 Destructors . . . 299
12.808 so_45071876_45072033_0 . . . 299
12.809 so_8137904_8138772_1 . . . 300
12.810 so_24000710_24000744_3 . . . 300
12.811 so_29520961_29521080_0 . . . 300
12.812 so_2271046_2271055_1 . . . 300
12.813 so_35383453_35383576_5 . . . 300
12.814 so_10756354_10756554_0 . . . 300
12.815 so_16540725_16540757_1 . . . 300
12.816 so_26331364_26332289_0 . . . 300
12.817 so_27737492_27737913_0 . . . 300
12.818 so_17177386_17177386_0 . . . 300
12.819 so_14924827_14925081_0 . . . 300
12.820 so_26395476_26395492_2 . . . 300
12.821 so_36760797_36760987_0 . . . 300
12.822 so_6273958_6274011_1 . . . 300
12.823 so_27050187_27050336_0 . . . 300
12.824 so_42325978_42326111_0 . . . 300
12.825 so_13771823_13771838_1 . . . 300
12.826 so_31482306_31482517_1 . . . 300
12.827 so_29861818_29862016_0 . . . 300
12.828 so_33080528_33080560_0 . . . 300
12.829 so_18416780_18416814_0 . . . 300

12.830 so_24167652_24167902_3 . . . 300
12.831 so_24167652_24167902_1 . . . 300
12.832 so_19606492_19607079_1 . . . 300
12.833 so_42416258_42418091_4 . . . 300
12.834 so_33254582_33255086_0 . . . 300
12.835 so_45071876_45072047_8 . . . 300
12.836 so_45071876_45072047_7 . . . 300
12.837 so_3109134_3109160_1 . . . 300
12.838 so_42325978_42326111_3 . . . 300
12.839 so_42325978_42326111_4 . . . 300
12.840 so_30096911_30097049_4 . . . 300
12.841 so_33626088_33627157_4 . . . 300
12.842 so_33626088_33627157_6 . . . 300
12.843 so_46402983_46403511_5 . . . 300
12.844 so_45628694_45628834_9 . . . 300
12.845 so_45628694_45628834_10 . . . 300
12.846 so_14566341_14566341_0 . . . 300
12.847 so_18667318_18669481_2 . . . 300
12.848 so_18667318_18669481_3 . . . 300
12.849 so_19191211_19191306_0 . . . 300
12.850 so_24167652_24167902_15 . . . 300
12.851 so_24167652_24167902_14 . . . 300
12.852 so_24167652_24167902_8 . . . 300
12.853 so_24167652_24167902_9 . . . 300
12.854 so_19606492_19607079_2 . . . 300
12.855 so_46484307_46484307_3 . . . 300
12.856 so_46484307_46484307_2 . . . 300
12.857 so_44886942_45944822_1 . . . 300
12.858 so_38103578_38104223_13 . . . 300
12.859 so_38103578_38104223_12 . . . 300
12.860 so_3291507_3291787_5 . . . 300
12.861 so_3291507_3291787_6 . . . 300
12.862 so_42416258_42418091_5 . . . 300
12.863 so_33254582_33255086_3 . . . 300
12.864 so_33254582_33255086_4 . . . 300
12.865 so_33254582_33255086_1 . . . 300
12.866 so_36760797_36760987_1 . . . 300
12.867 so_27050187_27050336_1 . . . 300
12.868 so_25193041_25193277_0 . . . 300
12.869 so_42325978_42326111_6 . . . 300
12.870 so_2371176_2371368_2 . . . 301
12.871 so_17093474_17093825_0 . . . 301
12.872 so_40403417_40403574_0 . . . 301
12.873 so_16778210_16778210_0 . . . 301
12.874 so_28529185_28529355_5 . . . 301
12.875 so_13988811_13989085_0 . . . 301
12.876 so_3261694_3261770_0 . . . 301
12.877 so_3261694_3261730_0 . . . 301
12.878 so_34966340_34966340_1 . . . 301
12.879 so_34966340_34966376_1 . . . 301
12.880 so_22872699_22872747_1 . . . 301

12.881 so_12764294_12764352_1 . . . 301
12.882 so_12764294_12764308_0 . . . 301
12.883 so_13771823_13771838_0 . . . 301
12.884 so_31971452_31972038_0 . . . 301
12.885 so_37273501_37273812_0 . . . 301
12.886 so_28356069_28356248_0 . . . 301
12.887 so_20443257_20443340_0 . . . 301
12.888 so_29861818_29862016_2 . . . 301
12.889 so_3063341_3063341_1 . . . 301
12.890 so_4172722_4172724_3 . . . 301
12.891 so_33080528_33080560_1 . . . 301
12.892 so_3233078_3233081_3 . . . 301
12.893 so_13019129_13019364_4 . . . 301
12.894 so_28928590_28929939_0 . . . 301
12.895 so_33210342_33210413_1 . . . 301
12.896 so_14490877_14490963_0 . . . 301
12.897 so_28284740_28284903_1 . . . 301
12.898 so_17177386_17178402_2 . . . 301
12.899 so_31791103_31791236_4 . . . 301
12.900 so_25997329_25997527_5 . . . 301
12.901 so_25997329_25997527_6 . . . 301
12.902 so_13285694_13285777_0 . . . 301
12.903 so_40106941_40107282_2 . . . 301
12.904 so_5952837_5955153_0 . . . 301
12.905 so_22875320_22897555_1 . . . 301
12.906 so_26557830_26557853_0 . . . 301
12.907 so_31518581_31519337_0 . . . 301
12.908 so_25639140_25639304_0 . . . 301
12.909 so_36760797_36760987_2 . . . 301
12.910 so_10962372_17998615_1 . . . 301
12.911 so_12592865_17998482_1 . . . 301
12.912 so_24229131_24229131_1 . . . 301
12.913 so_12918637_12918985_0 . . . 301
12.914 so_24854236_24855566_0 . . . 301
12.915 so_37777606_37778582_0 . . . 301
12.916 so_999340_1000176_0 . . . 301
12.917 so_30392041_30394552_0 . . . 301
12.918 so_31791103_31791236_10 . . . 301
12.919 so_3199995_3200128_0 . . . 301
12.920 so_41779374_41779932_1 . . . 301
12.921 so_20927795_20927819_0 . . . 301
12.922 so_27739540_27739653_2 . . . 301
12.923 so_42325978_42326111_10 . . . 301
12.924 so_42325978_42326111_8 . . . 301
12.925 so_42325978_42326111_9 . . . 301
12.926 so_42325978_42326111_11 . . . 301
12.927 so_26036384_26045084_3 . . . 301
12.928 so_33498101_33511384_1 . . . 301
12.929 so_19790804_19791281_9 . . . 301
12.930 so_19790804_19791281_4 . . . 301
12.931 so_19790804_19791281_8 . . . 301

12.932 so_19790804_19791281_5 . . . 301
12.933 so_19790804_19791281_6 . . . 301
12.934 so_19790804_19791281_10 . . . 301
12.935 so_29861818_29862016_1 . . . 301
12.936 so_29448997_29449097_0 . . . 301
12.937 so_7396059_7396086_1 . . . 301
12.938 so_26986570_26995474_0 . . . 301
12.939 so_45628694_45628896_4 . . . 301
12.940 so_45628694_45628896_3 . . . 301
12.941 so_45628694_45628896_1 . . . 301
12.942 so_45628694_45628896_2 . . . 301
12.943 so_45628694_45628896_5 . . . 301
12.944 so_6979612_6980506_1 . . . 301
12.945 so_2087600_2087621_0 . . . 301
12.946 so_28085847_28087336_4 . . . 301
12.947 so_42793159_42793281_2 . . . 301
12.948 so_35383453_35383576_6 . . . 301
12.949 so_47559033_47565432_2 . . . 301
12.950 so_2196327_2196495_11 . . . 301
12.951 so_8047743_8047778_0 . . . 301
12.952 so_36927212_36927743_0 . . . 301
12.953 so_31498758_31498758_0 . . . 301
12.954 so_47278993_47279278_1 . . . 301
12.955 so_13897394_13897423_0 . . . 301
12.956 so_33866858_33868842_0 . . . 301
12.957 so_2693199_2693199_3 . . . 301
12.958 so_22875320_22897555_2 . . . 301
12.959 so_39162251_39162585_2 . . . 301
12.960 so_7396059_7396086_2 . . . 302
12.961 so_42793159_42793281_4 . . . 302
12.962 so_40403417_40403682_3 . . . 302
12.963 so_24771148_24771182_0 . . . 302
12.964 so_32928787_32929099_1 . . . 302
12.965 so_31505399_31506285_1 . . . 302
12.966 so_3063341_3063341_0 . . . 302
12.967 so_3063341_3063401_3 . . . 302
12.968 so_23660498_23660634_0 . . . 302
12.969 so_30310299_30310501_0 . . . 302
12.970 so_7396059_7396086_3 . . . 302
12.971 so_28928590_28930230_1 . . . 302
12.972 so_30466949_30468523_1 . . . 302
12.973 so_21547180_21547181_1 . . . 302
12.974 so_30315524_30315524_0 . . . 302
12.975 so_2496902_2496917_0 . . . 302
12.976 so_24047340_24048582_1 . . . 302
12.977 so_11235357_11235479_0 . . . 302
12.978 so_46647004_46647212_0 . . . 302
12.979 so_42793159_42793281_3 . . . 302
12.980 so_35744832_35745378_7 . . . 302
12.981 so_3291507_3291787_0 . . . 302
12.982 so_35383453_35383576_7 . . . 302

12.983 so_22026587_22026701_0 . . . 302
12.984 so_8982575_8982624_0 . . . 302
12.985 so_34424726_34425285_2 . . . 302
12.986 so_456310_456339_0 . . . 302
12.987 so_10858998_10859012_0 . . . 302
12.988 Free store . . . 303
12.989 so_5406199_5406262_0 . . . 303
12.990 so_25875228_25875313_1 . . . 303
12.991 so_13897394_13897421_0 . . . 303
12.992 so_17609609_17609653_0 . . . 303
12.993 so_44338559_44338825_3 . . . 303
12.994 so_6355438_6355438_0 . . . 303
12.995 so_11745527_11747027_0 . . . 303
12.996 so_16100069_16100415_3 . . . 303
12.997 so_14730896_14732625_2 . . . 303
12.998 so_14730896_14732625_1 . . . 303
12.999 so_45041045_45052333_10 . . . 303
12.1000 so_7360752_11814437_6 . . . 303
12.1001 so_47278993_47279278_0 . . . 303
12.1002 so_25121295_25121986_0 . . . 303
12.1003 so_34879858_34912703_4 . . . 303
12.1004 so_34879858_34912703_3 . . . 303
12.1005 so_45256524_45256525_0 . . . 303
12.1006 so_33461724_33461842_0 . . . 303
12.1007 so_13897394_13897423_1 . . . 303
12.1008 so_13897394_13897421_1 . . . 303
12.1009 so_6355438_6355438_1 . . . 304
12.1010 so_6355438_6355438_2 . . . 304
12.1011 Initialization . . . 304
12.1012 so_7411515_7411708_0 . . . 304
12.1013 so_24702484_24702754_1 . . . 304
12.1014 so_11965214_11965258_0 . . . 304
12.1015 so_23720258_23720289_3 . . . 304
12.1016 so_23720258_23720289_4 . . . 305
12.1017 so_19030789_19030839_0 . . . 305
12.1018 so_23720258_23720289_5 . . . 305
12.1019 so_31101854_31102112_1 . . . 305
12.1020 so_19904794_19905083_0 . . . 305
12.1021 so_17477573_17477573_0 . . . 305
12.1022 so_13565742_13565871_0 . . . 305
12.1023 so_24702484_24702754_3 . . . 305
12.1024 so_23720258_23720289_6 . . . 305
12.1025 so_32835803_32835803_0 . . . 305
12.1026 so_24574092_24574182_0 . . . 306
12.1027 so_16288824_16288955_0 . . . 306
12.1028 so_28673545_28673797_0 . . . 306
12.1029 so_5497987_5498038_1 . . . 306
12.1030 so_22647524_22647552_0 . . . 306
12.1031 so_27317232_27317821_1 . . . 306
12.1032 so_18278365_18278467_1 . . . 306
12.1033 so_18574882_18574969_1 . . . 306

12.1034 so_20615156_20618028_0 . . . 306
12.1035 so_23598199_23598296_0 . . . 306
12.1036 so_38886035_38887869_0 . . . 306
12.1037 so_28673545_28673758_1 . . . 306
12.1038 so_28673545_28673545_0 . . . 306
12.1039 so_3878670_3878862_0 . . . 306
12.1040 so_25816353_25816353_0 . . . 306
12.1041 so_41148054_41195598_0 . . . 306
12.1042 so_28932026_28932026_0 . . . 306
12.1043 so_7568611_7568637_0 . . . 306
12.1044 so_43954667_43954745_0 . . . 306
12.1045 so_2290733_2290803_1 . . . 306
12.1046 so_15441294_15441379_0 . . . 306
12.1047 so_8887864_8887980_0 . . . 306
12.1048 so_23493598_23493639_0 . . . 306
12.1049 so_10693913_10694924_1 . . . 307
12.1050 so_28673545_28673758_2 . . . 307
12.1051 so_28673545_28673758_3 . . . 307
12.1052 so_28673545_28673746_0 . . . 307
12.1053 so_30686129_30686175_0 . . . 307
12.1054 so_25348530_25348530_1 . . . 307
12.1055 so_25348530_25348530_0 . . . 307
12.1056 so_25348530_25348602_0 . . . 307
12.1057 so_30674317_30675023_1 . . . 307
12.1058 so_28772610_28772717_1 . . . 307
12.1059 so_33115803_33116008_1 . . . 307
12.1060 so_25375833_25375833_0 . . . 307
12.1061 so_38486235_38486324_0 . . . 307
12.1062 so_12190051_12190126_0 . . . 307
12.1063 so_38426461_38426597_3 . . . 307
12.1064 so_16288824_16288955_1 . . . 307
12.1065 so_37668952_37674591_2 . . . 307
12.1066 so_26914076_36884406_1 . . . 307
12.1067 so_27671276_27671997_0 . . . 307
12.1068 so_7423008_7423387_1 . . . 307
12.1069 so_14259602_14259765_0 . . . 307
12.1070 so_32016483_32016834_1 . . . 307
12.1071 so_19564082_19564267_0 . . . 307
12.1072 so_39124246_39124394_2 . . . 307
12.1073 so_20849454_20849500_0 . . . 307
12.1074 so_21225086_21225154_1 . . . 307
12.1075 so_24526475_32214536_9 . . . 307
12.1076 so_22329703_22330014_0 . . . 307
12.1077 so_22329703_22329975_0 . . . 307
12.1078 so_43479839_43480231_2 . . . 307
12.1079 so_26098900_26099873_0 . . . 307
12.1080 so_34993429_34995780_1 . . . 307
12.1081 so_23493598_23493639_1 . . . 307
12.1082 so_18997929_18999167_2 . . . 307
12.1083 so_27352021_27354688_0 . . . 307
12.1084 so_2371176_2371368_0 . . . 308

12.1085 so_38426461_38426597_2 . . . 308
12.1086 so_38426461_38426597_1 . . . 308
12.1087 so_31388208_31390380_5 . . . 308
12.1088 so_31388208_31390380_3 . . . 308
12.1089 so_31388208_31390380_1 . . . 308
12.1090 so_31388208_31390380_7 . . . 308
12.1091 so_31388208_31390380_2 . . . 308
12.1092 so_25397384_25397384_0 . . . 308
12.1093 so_32961635_32961635_0 . . . 308
12.1094 so_24513692_24514151_0 . . . 308
12.1095 so_26182734_26183427_6 . . . 308
12.1096 so_26182734_26183427_5 . . . 308
12.1097 so_25501583_25501734_0 . . . 308
12.1098 so_4438689_4443548_0 . . . 308
12.1099 so_35886358_35886494_0 . . . 308
12.1100 so_35886358_35886494_1 . . . 308
12.1101 so_17905265_17905430_0 . . . 308
12.1102 so_17905265_17905422_0 . . . 308
12.1103 so_40307941_40309457_0 . . . 308
12.1104 so_11693512_11694039_0 . . . 308
12.1105 so_24943665_24943751_2 . . . 308
12.1106 so_24943665_24943751_3 . . . 308
12.1107 so_4502728_4503088_0 . . . 308
12.1108 so_20806185_20806206_1 . . . 308
12.1109 so_7760291_7760316_4 . . . 308
12.1110 so_13662441_13662441_1 . . . 308
12.1111 so_4384052_4384052_0 . . . 308
12.1112 so_4384052_4384064_0 . . . 308
12.1113 so_44094458_44094815_2 . . . 308
12.1114 so_44094458_44094815_0 . . . 308
12.1115 so_44094458_44094815_3 . . . 308
12.1116 so_44094458_44094815_5 . . . 308
12.1117 so_24713833_24714285_0 . . . 308
12.1118 so_33178981_33179745_0 . . . 308
12.1119 so_33178981_33179745_3 . . . 308
12.1120 so_33178981_33179745_2 . . . 308
12.1121 so_33178981_33179745_4 . . . 308
12.1122 so_33178981_33179745_1 . . . 308
12.1123 so_30992462_30992563_3 . . . 308
12.1124 so_30992462_30992563_0 . . . 308
12.1125 so_30992462_30992563_1 . . . 308
12.1126 so_47437349_47447101_0 . . . 308
12.1127 so_24537075_24537466_0 . . . 308
12.1128 so_25278066_25278130_1 . . . 308
12.1129 so_25278066_25278130_0 . . . 308
12.1130 so_25278066_25278130_2 . . . 308
12.1131 so_28221245_28221486_1 . . . 308
12.1132 so_28221245_28221486_0 . . . 308
12.1133 so_21225086_21225154_0 . . . 308
12.1134 so_15212261_15212447_5 . . . 308
12.1135 so_15212261_15212447_2 . . . 308

12.1136 so_15212261_15212447_3 . . . 308
12.1137 so_38447455_38447623_0 . . . 308
12.1138 so_39057240_39057301_0 . . . 308
12.1139 so_39057240_39057240_0 . . . 308
12.1140 so_24526475_32214536_12 . . . 308
12.1141 so_24526475_32214536_15 . . . 308
12.1142 so_24526475_32214536_11 . . . 308
12.1143 so_24526475_32214536_13 . . . 308
12.1144 so_25375833_25375833_1 . . . 308
12.1145 so_40932844_40933634_1 . . . 308
12.1146 so_40932844_40933634_2 . . . 308
12.1147 so_34856298_34856453_0 . . . 308
12.1148 so_34856298_34856453_1 . . . 308
12.1149 so_10917875_10917935_2 . . . 308
12.1150 so_10917875_10917935_1 . . . 308
12.1151 so_29593207_29593683_0 . . . 308
12.1152 so_43703229_43839226_2 . . . 308
12.1153 so_34071187_34071365_2 . . . 308
12.1154 so_34071187_34071365_3 . . . 308
12.1155 so_16782103_16782131_0 . . . 308
12.1156 so_15465122_15465181_3 . . . 308
12.1157 so_15465122_15465181_1 . . . 308
12.1158 so_15465122_15465181_2 . . . 308
12.1159 so_26098900_26099873_1 . . . 308
12.1160 so_28343407_28343506_0 . . . 308
12.1161 so_1144264_1144404_5 . . . 308
12.1162 so_1144264_1144404_4 . . . 308
12.1163 so_46427709_46430666_7 . . . 308
12.1164 so_46427709_46430666_4 . . . 308
12.1165 so_46427709_46430666_5 . . . 308
12.1166 so_46427709_46430666_6 . . . 308
12.1167 so_11637611_11637902_7 . . . 308
12.1168 so_18901315_18902462_5 . . . 308
12.1169 so_35383453_35383576_16 . . . 308
12.1170 so_14659752_14659931_5 . . . 308
12.1171 so_14659752_14659881_0 . . . 308
12.1172 so_27352021_27354688_2 . . . 308
12.1173 so_27352021_27354688_1 . . . 308
12.1174 so_21468642_21469852_0 . . . 308
12.1175 so_17790278_17790901_1 . . . 308
12.1176 so_26866859_26867518_0 . . . 308
12.1177 so_19877218_19877363_0 . . . 308
12.1178 so_19877218_19877363_1 . . . 308
12.1179 so_22991180_22991232_0 . . . 308
12.1180 so_22991180_22991232_1 . . . 308
12.1181 so_38762157_38764809_0 . . . 308
12.1182 so_20303525_20303724_0 . . . 308
12.1183 so_20303525_20303724_2 . . . 308
12.1184 so_40378205_40378370_3 . . . 308
12.1185 so_40378205_40378370_2 . . . 308
12.1186 so_39124246_39124394_1 . . . 308

12.1187 so_39124246_39124394_0 . . . 308
12.1188 so_29593207_29593683_2 . . . 308
12.1189 so_29593207_29593683_1 . . . 308
12.1190 so_23928080_23928387_0 . . . 308
12.1191 so_24226054_24226136_1 . . . 308
12.1192 so_24226054_24226136_0 . . . 308
12.1193 so_36600187_36601200_0 . . . 308
12.1194 so_36600187_36601200_1 . . . 308
12.1195 so_14886104_14886228_1 . . . 308
12.1196 so_14886104_14886228_0 . . . 308
12.1197 so_8218397_8218460_1 . . . 308
12.1198 so_34690128_34690286_1 . . . 309
12.1199 so_27739540_27739653_1 . . . 309
12.1200 so_26036384_26045084_2 . . . 309
12.1201 so_33498101_33511384_0 . . . 309
12.1202 so_43680035_43680348_2 . . . 309
12.1203 so_27336779_27624369_0 . . . 309
12.1204 so_26986570_26995474_1 . . . 309
12.1205 so_26986570_26995870_0 . . . 309
12.1206 so_47559033_47565432_1 . . . 309
12.1207 so_6308915_6308944_0 . . . 309
12.1208 so_16288824_16288955_2 . . . 309
12.1209 so_24577441_24577525_0 . . . 309
12.1210 so_30144345_30144441_0 . . . 309
12.1211 so_25852217_25852390_3 . . . 309
12.1212 so_23109393_23109589_1 . . . 309
12.1213 so_23109393_23109589_4 . . . 309
12.1214 so_23109393_23109589_2 . . . 309
12.1215 so_23109393_23109589_0 . . . 309
12.1216 so_23109393_23109589_3 . . . 309
12.1217 so_39373556_39379584_0 . . . 309
12.1218 so_27457101_27457219_0 . . . 309
12.1219 so_27457101_27457219_1 . . . 309
12.1220 so_27457101_27457219_3 . . . 309
12.1221 so_27457101_27457219_2 . . . 309
12.1222 so_27457101_27457219_4 . . . 309
12.1223 so_37668952_37674591_1 . . . 309
12.1224 so_37668952_37674591_0 . . . 309
12.1225 so_44863382_44863566_1 . . . 309
12.1226 so_16033729_16033892_3 . . . 309
12.1227 so_16033729_16033892_2 . . . 309
12.1228 so_16033729_16033892_4 . . . 309
12.1229 so_16033729_16033892_5 . . . 309
12.1230 so_16033729_16033892_1 . . . 309
12.1231 so_16033729_16033892_6 . . . 309
12.1232 so_11181317_11181418_3 . . . 309
12.1233 so_11181317_11181418_5 . . . 309
12.1234 so_11181317_11181418_1 . . . 309
12.1235 so_11181317_11181418_2 . . . 309
12.1236 so_11181317_11181418_4 . . . 309
12.1237 so_11181317_11181418_0 . . . 309

12.1238 so_31190982_31191072_0 . . . 309
12.1239 so_19370281_19372824_1 . . . 309
12.1240 so_19370281_19372824_2 . . . 309
12.1241 so_19370281_19372824_0 . . . 309
12.1242 so_19370281_19372824_3 . . . 309
12.1243 so_19370281_19372824_4 . . . 309
12.1244 so_41853440_41853668_6 . . . 309
12.1245 so_41853440_41853668_4 . . . 309
12.1246 so_41853440_41853668_7 . . . 309
12.1247 so_41853440_41853668_3 . . . 309
12.1248 so_41853440_41853668_5 . . . 309
12.1249 so_41853440_41853668_2 . . . 309
12.1250 so_6286656_6286739_0 . . . 309
12.1251 so_20409500_20409911_0 . . . 309
12.1252 so_32934179_32934226_1 . . . 309
12.1253 so_3608108_3608208_2 . . . 309
12.1254 so_3608108_3608208_3 . . . 309
12.1255 so_3608108_3608208_4 . . . 309
12.1256 so_3608108_3608208_5 . . . 309
12.1257 so_28529185_28529355_3 . . . 309
12.1258 so_28529185_28529355_1 . . . 309
12.1259 so_28529185_28529355_4 . . . 309
12.1260 so_28529185_28529355_0 . . . 309
12.1261 so_28529185_28529355_2 . . . 309
12.1262 so_6515042_6515176_0 . . . 309
12.1263 so_4438689_4438738_0 . . . 309
12.1264 so_12285638_12285672_1 . . . 309
12.1265 so_12285638_12285672_0 . . . 309
12.1266 so_24281500_24281583_0 . . . 309
12.1267 so_24281500_24281583_1 . . . 309
12.1268 so_43649375_44171445_3 . . . 309
12.1269 so_17905265_17905422_1 . . . 309
12.1270 so_2254263_2254306_4 . . . 309
12.1271 so_2254263_2254306_2 . . . 309
12.1272 so_2254263_2254306_5 . . . 309
12.1273 so_2254263_2254306_0 . . . 309
12.1274 so_2254263_2254306_3 . . . 309
12.1275 so_12764294_12764352_0 . . . 309
12.1276 so_16945465_16946035_0 . . . 309
12.1277 so_22642699_22643080_2 . . . 309
12.1278 so_22642699_22643080_3 . . . 309
12.1279 so_22642699_22643080_1 . . . 309
12.1280 so_22642699_22643080_5 . . . 309
12.1281 so_22642699_22643080_4 . . . 309
12.1282 so_18929006_18929072_5 . . . 309
12.1283 so_38883526_38883621_4 . . . 309
12.1284 so_38883526_38883621_1 . . . 309
12.1285 so_38883526_38883621_3 . . . 309
12.1286 so_38883526_38883621_2 . . . 309
12.1287 so_38883526_38883621_0 . . . 309
12.1288 so_25033690_25033791_4 . . . 309

12.1289 so_25033690_25033791_1 . . . 309
12.1290 so_25033690_25033791_3 . . . 309
12.1291 so_25033690_25033791_2 . . . 309
12.1292 so_19395232_19395316_0 . . . 309
12.1293 so_20443257_20443340_1 . . . 309
12.1294 so_20443257_20443340_3 . . . 309
12.1295 so_20443257_20443340_2 . . . 309
12.1296 so_40633206_40633475_3 . . . 309
12.1297 so_40633206_40633475_1 . . . 309
12.1298 so_40633206_40633475_2 . . . 309
12.1299 so_40633206_40633475_4 . . . 309
12.1300 so_40633206_40633475_0 . . . 309
12.1301 so_40633206_40633475_5 . . . 309
12.1302 so_8346400_8346843_1 . . . 309
12.1303 so_8346400_8346843_2 . . . 309
12.1304 so_8346400_8346843_3 . . . 309
12.1305 so_45835809_45835853_3 . . . 309
12.1306 so_45835809_45835855_2 . . . 309
12.1307 so_45835809_45835855_1 . . . 309
12.1308 so_45835809_45835853_4 . . . 309
12.1309 so_45835809_45835853_2 . . . 309
12.1310 so_17125596_17125650_6 . . . 309
12.1311 so_17125596_17125650_3 . . . 309
12.1312 so_17125596_17125650_2 . . . 309
12.1313 so_17125596_17125650_4 . . . 309
12.1314 so_17125596_17125650_5 . . . 309
12.1315 so_26076997_26077091_6 . . . 309
12.1316 so_26076997_26077091_3 . . . 309
12.1317 so_26076997_26077091_4 . . . 309
12.1318 so_26076997_26077091_1 . . . 309
12.1319 so_26076997_26077091_5 . . . 309
12.1320 so_26076997_26077091_2 . . . 309
12.1321 so_3507100_3510583_1 . . . 309
12.1322 so_17804293_17805437_0 . . . 309
12.1323 so_30131370_30131485_0 . . . 309
12.1324 so_44094458_44094458_3 . . . 309
12.1325 so_44094458_44094458_1 . . . 309
12.1326 so_44094458_44094458_2 . . . 309
12.1327 so_23875523_23875561_1 . . . 309
12.1328 so_23875523_23875561_0 . . . 309
12.1329 so_23875523_23875561_3 . . . 309
12.1330 so_23875523_23875561_5 . . . 309
12.1331 so_23875523_23875561_2 . . . 309
12.1332 so_23875523_23875561_4 . . . 309
12.1333 so_12793152_12793512_0 . . . 309
12.1334 so_12793152_12793512_1 . . . 309
12.1335 so_7086445_7086498_3 . . . 309
12.1336 so_31642431_31642806_2 . . . 309
12.1337 so_31642431_31642806_3 . . . 309
12.1338 so_31642431_31642806_1 . . . 309
12.1339 so_16873215_16873237_4 . . . 309

12.1340 so_16873215_16873237_2 . . . 309
12.1341 so_16873215_16873237_1 . . . 309
12.1342 so_16873215_16873237_5 . . . 309
12.1343 so_16873215_16873237_3 . . . 309
12.1344 so_30671157_30671218_1 . . . 309
12.1345 so_30671157_30671218_0 . . . 309
12.1346 so_28928590_28929939_1 . . . 309
12.1347 so_31798217_31798261_1 . . . 309
12.1348 so_31798217_31798261_2 . . . 309
12.1349 so_31798217_31798261_3 . . . 309
12.1350 so_31798217_31798261_0 . . . 309
12.1351 so_47469805_47470669_2 . . . 309
12.1352 so_47469805_47470669_3 . . . 309
12.1353 so_25278066_25278189_0 . . . 309
12.1354 so_24285112_24287946_1 . . . 309
12.1355 so_24285112_24287946_2 . . . 309
12.1356 so_24285112_24287946_0 . . . 309
12.1357 so_21033845_21034121_1 . . . 309
12.1358 so_21033845_21034140_0 . . . 309
12.1359 so_21033845_21034121_0 . . . 309
12.1360 so_35335167_35335368_0 . . . 309
12.1361 so_15366621_15366747_4 . . . 309
12.1362 so_15366621_15366747_1 . . . 309
12.1363 so_15366621_15366747_0 . . . 309
12.1364 so_15366621_15366747_3 . . . 309
12.1365 so_15366621_15366747_2 . . . 309
12.1366 so_30942132_30942227_2 . . . 309
12.1367 so_30942132_30942227_0 . . . 309
12.1368 so_30942132_30942227_3 . . . 309
12.1369 so_30942132_30942227_1 . . . 309
12.1370 so_30942132_30942227_4 . . . 309
12.1371 so_26007869_26008040_0 . . . 309
12.1372 so_25186264_25186641_0 . . . 309
12.1373 so_4037219_4037227_0 . . . 309
12.1374 so_24526475_32214536_18 . . . 309
12.1375 so_24526475_32214536_19 . . . 309
12.1376 so_24526475_32214536_22 . . . 309
12.1377 so_24526475_32214536_20 . . . 309
12.1378 so_24526475_32214536_21 . . . 309
12.1379 so_1242830_38990055_0 . . . 309
12.1380 so_18479295_18479406_5 . . . 309
12.1381 so_18479295_18479406_1 . . . 309
12.1382 so_18479295_18479406_2 . . . 309
12.1383 so_18479295_18479406_4 . . . 309
12.1384 so_18479295_18479406_3 . . . 309
12.1385 so_18479295_18479406_0 . . . 309
12.1386 so_11311732_11312376_3 . . . 309
12.1387 so_11311732_11312356_0 . . . 309
12.1388 so_11311732_11312376_2 . . . 309
12.1389 so_11311732_11312376_0 . . . 309
12.1390 so_23928080_23928387_1 . . . 309

12.1391 so_43703229_43703341_1 . . . 309
12.1392 so_10720377_10720417_0 . . . 309
12.1393 so_40991729_40992205_2 . . . 309
12.1394 so_35314297_35314557_1 . . . 309
12.1395 so_35314297_35314557_0 . . . 309
12.1396 so_35314297_35314557_2 . . . 309
12.1397 so_33574233_33574389_0 . . . 309
12.1398 so_10304300_10304523_1 . . . 309
12.1399 so_10304300_10304523_3 . . . 309
12.1400 so_10304300_10304523_2 . . . 309
12.1401 so_10304300_10304523_4 . . . 309
12.1402 so_10304300_10304523_0 . . . 309
12.1403 so_1774187_1774213_2 . . . 309
12.1404 so_1774187_1774213_3 . . . 309
12.1405 so_1774187_1774213_1 . . . 309
12.1406 so_25997329_25997527_3 . . . 309
12.1407 so_25997329_25997527_4 . . . 309
12.1408 so_25997329_25997527_1 . . . 309
12.1409 so_25997329_25997527_0 . . . 309
12.1410 so_25997329_25997527_2 . . . 309
12.1411 so_30258639_30259298_8 . . . 309
12.1412 so_30258639_30259298_12 . . . 309
12.1413 so_30258639_30259298_9 . . . 309
12.1414 so_30258639_30259298_11 . . . 309
12.1415 so_30258639_30259298_10 . . . 309
12.1416 so_43479839_43480231_1 . . . 309
12.1417 so_34993429_34995780_0 . . . 309
12.1418 so_13863219_13882851_1 . . . 309
12.1419 so_30744416_30744619_1 . . . 309
12.1420 so_46267174_46267692_3 . . . 309
12.1421 so_46267174_46267692_5 . . . 309
12.1422 so_46267174_46267692_1 . . . 309
12.1423 so_46267174_46267692_4 . . . 309
12.1424 so_46267174_46267692_2 . . . 309
12.1425 so_38211185_38211406_0 . . . 309
12.1426 so_38211185_38211406_1 . . . 309
12.1427 so_31518581_31519337_1 . . . 309
12.1428 so_7405839_7405897_0 . . . 309
12.1429 so_7405839_7405874_1 . . . 309
12.1430 so_7405839_7405874_3 . . . 309
12.1431 so_7405839_7405874_2 . . . 309
12.1432 so_30350127_30350754_1 . . . 309
12.1433 so_4417544_4417604_3 . . . 309
12.1434 so_4417544_4417604_4 . . . 309
12.1435 so_4417544_4417604_1 . . . 309
12.1436 so_2669888_2669911_3 . . . 309
12.1437 so_2669888_2669911_2 . . . 309
12.1438 so_2669888_2669911_1 . . . 309
12.1439 so_2669888_2669911_4 . . . 309
12.1440 so_18997929_18999167_0 . . . 309
12.1441 so_18997929_18999167_1 . . . 309

12.1442 so_8854354_8854435_1 . . . 309
12.1443 so_14583688_14583866_0 . . . 309
12.1444 so_20615156_20618028_1 . . . 310
12.1445 so_11283977_12429723_0 . . . 310
12.1446 so_2227244_2227278_0 . . . 310
12.1447 so_1193134_1193691_0 . . . 310
12.1448 so_11333671_11333795_1 . . . 310
12.1449 so_11333671_11333795_2 . . . 310
12.1450 so_14687396_14687396_1 . . . 310
12.1451 so_12793152_12793512_3 . . . 310
12.1452 so_12793152_12793512_2 . . . 310
12.1453 so_4006160_4006252_0 . . . 310
12.1454 so_4006160_4006252_2 . . . 310
12.1455 so_4006160_4006252_3 . . . 310
12.1456 so_15751618_15752887_0 . . . 310
12.1457 so_6932243_6932435_0 . . . 310
12.1458 so_43479839_43480189_1 . . . 310
12.1459 so_6185020_6185043_2 . . . 310
12.1460 so_6185020_6185043_3 . . . 310
12.1461 so_6185020_6185105_0 . . . 310
12.1462 so_6185020_6185043_1 . . . 310
12.1463 so_43954667_43954745_2 . . . 310
12.1464 so_43954667_43954745_1 . . . 310
12.1465 so_22832001_22832370_1 . . . 310
12.1466 so_22832001_22832370_2 . . . 310
12.1467 so_30144345_30144949_0 . . . 310
12.1468 so_46488260_46488260_0 . . . 310
12.1469 so_41853440_41853668_0 . . . 310
12.1470 so_30438446_30438515_0 . . . 310
12.1471 so_11174129_11174627_0 . . . 310
12.1472 so_6591859_6591952_0 . . . 310
12.1473 so_15751618_15752887_1 . . . 310
12.1474 so_25754796_25755113_0 . . . 310
12.1475 so_11335525_11335683_0 . . . 310
12.1476 so_11335525_11335683_1 . . . 310
12.1477 so_30826000_30826049_0 . . . 310
12.1478 so_2492318_2492395_0 . . . 310
12.1479 so_14681349_14681349_0 . . . 310
12.1480 so_30350127_30350754_2 . . . 310
12.1481 so_30780171_30780421_0 . . . 310
12.1482 so_29991719_29991820_1 . . . 310
12.1483 Construction and destruction . . . 311
12.1484 so_25470791_25470791_0 . . . 311
12.1485 so_2328960_2328970_0 . . . 311
12.1486 so_40403417_40403682_1 . . . 311
12.1487 so_11174129_11174627_1 . . . 311
12.1488 so_4502728_4502807_0 . . . 311
12.1489 so_45902829_45905157_2 . . . 311
12.1490 so_26704728_26704942_0 . . . 311
12.1491 so_25600845_25600939_0 . . . 311
12.1492 so_23357236_23365730_0 . . . 311

12.1493 so_23357236_23365730_1 . . . 311
12.1494 so_37409590_37411089_0 . . . 311
12.1495 so_7086445_7086498_1 . . . 311
12.1496 so_32940847_32945791_0 . . . 311
12.1497 so_40577808_40577869_3 . . . 311
12.1498 so_8611468_8611468_1 . . . 311
12.1499 so_33574112_33574286_1 . . . 311
12.1500 so_30020090_30020194_0 . . . 311
12.1501 so_46181083_46181498_0 . . . 311
12.1502 so_31493761_31496161_0 . . . 311
12.1503 so_29809241_29809865_0 . . . 311
12.1504 so_11637611_11637902_0 . . . 311
12.1505 so_35383453_35383576_8 . . . 311
12.1506 so_37644977_37646022_0 . . . 311
12.1507 so_7397069_7397317_2 . . . 311
12.1508 so_29821811_29821811_0 . . . 311
12.1509 so_29821811_29822863_2 . . . 311
12.1510 so_4267363_4357616_0 . . . 311
12.1511 so_4267363_4357616_1 . . . 311
12.1512 so_6299266_6299463_0 . . . 311
12.1513 so_37960088_37960952_4 . . . 311
12.1514 so_6435521_6436660_0 . . . 311
12.1515 so_33649370_33651899_11 . . . 311
12.1516 so_24194076_24194590_6 . . . 311
12.1517 so_8611468_8611504_1 . . . 311
12.1518 so_17116040_17116124_0 . . . 311
12.1519 so_23415661_23415662_12 . . . 311
12.1520 so_30258639_30259298_7 . . . 311
12.1521 so_11637611_11637902_1 . . . 311
12.1522 so_38487640_38488280_4 . . . 311
12.1523 so_21968982_21969764_0 . . . 312
12.1524 so_14444251_14444551_3 . . . 312
12.1525 so_4267363_4357616_2 . . . 312
12.1526 so_36843778_36846879_0 . . . 312
12.1527 so_7750280_7751040_0 . . . 312
12.1528 so_962132_45832805_0 . . . 312
12.1529 so_41732051_41732104_1 . . . 312
12.1530 so_41732051_41732104_0 . . . 312
12.1531 so_33749603_33749655_0 . . . 312
12.1532 so_18456695_18456764_0 . . . 312
12.1533 so_18456695_18457039_0 . . . 312
12.1534 so_37960088_37960952_5 . . . 312
12.1535 so_16719460_16719531_1 . . . 312
12.1536 so_11377629_11377629_0 . . . 312
12.1537 so_11377629_11377756_0 . . . 312
12.1538 so_25562042_25562868_2 . . . 312
12.1539 so_25562042_25562868_1 . . . 312
12.1540 so_25562042_25562042_1 . . . 312
12.1541 so_25562042_25562042_0 . . . 312
12.1542 so_25562042_25562868_0 . . . 312
12.1543 so_30882420_30882420_0 . . . 312

12.1544 so_33526666_33526987_0 . . . 312
12.1545 so_7473106_7473271_0 . . . 312
12.1546 so_7086445_7086498_0 . . . 312
12.1547 so_6030763_6030880_0 . . . 312
12.1548 so_6591859_6591952_1 . . . 312
12.1549 so_25343085_25343150_0 . . . 312
12.1550 so_10820974_10821000_0 . . . 312
12.1551 so_24500386_24500386_0 . . . 312
12.1552 so_24500386_24500476_0 . . . 312
12.1553 so_46485477_46485531_0 . . . 312
12.1554 so_11235357_11235427_0 . . . 312
12.1555 so_47307620_47308001_0 . . . 312
12.1556 so_37731137_37731335_0 . . . 312
12.1557 so_39389157_39389744_0 . . . 312
12.1558 so_30258639_30259298_5 . . . 312
12.1559 so_31129484_31129679_0 . . . 312
12.1560 so_30373_293047_1 . . . 312
12.1561 so_25541010_25541090_0 . . . 312
12.1562 so_25541010_25541090_3 . . . 312
12.1563 so_25541010_25541090_2 . . . 312
12.1564 so_25541010_25541010_4 . . . 312
12.1565 so_18456450_18456594_0 . . . 312
12.1566 so_21151264_21151590_0 . . . 312
12.1567 so_14549489_14550027_0 . . . 312
12.1568 so_14549489_14550864_0 . . . 312
12.1569 so_25723345_25723454_0 . . . 312
12.1570 so_7397069_7397317_1 . . . 312
12.1571 so_7397069_7397317_0 . . . 312
12.1572 so_45105150_45105150_0 . . . 313
12.1573 so_45103708_45103897_0 . . . 313
12.1574 so_21151264_21151590_1 . . . 313
12.1575 so_21206359_21206750_0 . . . 313
12.1576 so_30258639_30259298_6 . . . 313
12.1577 Copying and moving class objects . . . 314
12.1578 so_3021333_3022238_0 . . . 314
12.1579 so_32094881_32095069_0 . . . 314
12.1580 so_16984353_16984714_1 . . . 314
12.1581 so_16984353_16984714_0 . . . 314
12.1582 so_24238786_24239020_0 . . . 314
12.1583 so_30252074_30252333_0 . . . 314
12.1584 so_25360835_25360906_0 . . . 314
12.1585 so_11936730_11936784_0 . . . 314
12.1586 so_11028929_11029028_0 . . . 314
12.1587 so_22164789_22165202_1 . . . 314
12.1588 so_41184852_41184852_0 . . . 314
12.1589 so_39218428_39218464_0 . . . 314
12.1590 so_3281077_3281100_0 . . . 314
12.1591 so_4190585_4190689_0 . . . 314
12.1592 so_34746160_34749021_1 . . . 314
12.1593 so_12688942_12688958_0 . . . 314
12.1594 so_8436856_8436955_0 . . . 314

12.1595 so_8436856_8436953_0 . . . 314
12.1596 so_24925137_24925451_0 . . . 314
12.1597 so_41961993_41962227_0 . . . 314
12.1598 so_3648797_3648797_0 . . . 314
12.1599 so_32894131_32894445_1 . . . 314
12.1600 so_32894131_32895094_2 . . . 314
12.1601 so_32976919_32976919_0 . . . 314
12.1602 so_15115859_15116012_1 . . . 314
12.1603 so_33572861_33572962_0 . . . 314
12.1604 so_39023325_39023634_0 . . . 314
12.1605 so_2953611_2953925_1 . . . 314
12.1606 so_3527248_3527662_0 . . . 314
12.1607 so_22016901_22017381_0 . . . 314
12.1608 so_35382083_35382425_0 . . . 314
12.1609 so_14987847_14988094_2 . . . 314
12.1610 so_8143711_8143817_1 . . . 314
12.1611 so_38541369_38541434_0 . . . 314
12.1612 so_29966658_29966804_0 . . . 314
12.1613 so_24832143_24832271_0 . . . 314
12.1614 so_24832143_24832219_0 . . . 314
12.1615 so_10761342_10761380_4 . . . 314
12.1616 so_2787569_2787594_0 . . . 314
12.1617 so_27086214_27086214_0 . . . 314
12.1618 so_27898315_27898511_0 . . . 314
12.1619 so_11016262_11016403_0 . . . 314
12.1620 so_21210360_21210565_0 . . . 314
12.1621 so_17936009_17936135_0 . . . 314
12.1622 so_23830767_23831467_0 . . . 314
12.1623 so_19746474_19746828_0 . . . 314
12.1624 so_32381916_32382228_0 . . . 314
12.1625 so_7778698_7778720_0 . . . 314
12.1626 so_11028929_11029028_1 . . . 314
12.1627 so_26623158_26623158_0 . . . 314
12.1628 so_24488034_24488034_0 . . . 314
12.1629 so_24925137_24925451_1 . . . 314
12.1630 so_2953611_2953925_2 . . . 314
12.1631 so_38541369_38541434_2 . . . 314
12.1632 so_43481206_43481308_0 . . . 314
12.1633 so_34942627_34942954_0 . . . 314
12.1634 so_22164789_22165202_3 . . . 315
12.1635 so_7964360_7964428_2 . . . 315
12.1636 so_7964360_7964428_1 . . . 315
12.1637 so_41184852_41185071_0 . . . 315
12.1638 so_33532275_33532723_0 . . . 315
12.1639 so_23741529_23743884_1 . . . 315
12.1640 so_34746160_34749021_2 . . . 315
12.1641 so_8436856_8436955_1 . . . 315
12.1642 so_24925137_24925451_3 . . . 315
12.1643 so_3527248_3527662_1 . . . 315
12.1644 so_25833695_25833795_0 . . . 315
12.1645 so_29966658_29966804_1 . . . 315

12.1646 so_17780420_17780467_0 . . . 315
12.1647 so_2685854_2685975_0 . . . 315
12.1648 so_35546335_35546797_2 . . . 315
12.1649 so_23830767_23831467_3 . . . 315
12.1650 so_23830767_23831467_2 . . . 315
12.1651 so_11936730_11936784_1 . . . 315
12.1652 so_43954808_43954875_1 . . . 315
12.1653 so_5619292_5619475_0 . . . 315
12.1654 so_5619292_5623566_0 . . . 315
12.1655 so_29614410_29615191_1 . . . 315
12.1656 so_31571202_31571496_2 . . . 315
12.1657 so_22164789_22165202_5 . . . 315
12.1658 so_44553920_44554007_0 . . . 315
12.1659 so_25193041_25193277_1 . . . 315
12.1660 so_26954437_26954437_1 . . . 315
12.1661 so_7991666_7991832_1 . . . 315
12.1662 so_7991666_7991919_4 . . . 315
12.1663 so_11255027_11255089_0 . . . 315
12.1664 so_30120573_30120573_0 . . . 315
12.1665 so_10854948_10855018_0 . . . 315
12.1666 so_32235294_32235317_1 . . . 315
12.1667 so_12647437_12647564_0 . . . 315
12.1668 so_19267168_19267168_0 . . . 315
12.1669 so_6524185_6524855_0 . . . 315
12.1670 so_22468067_22468786_8 . . . 315
12.1671 so_35239552_35253265_1 . . . 315
12.1672 so_21013430_21013852_0 . . . 315
12.1673 so_21013430_21013943_0 . . . 315
12.1674 so_26667975_26667975_0 . . . 315
12.1675 so_25299622_25310189_2 . . . 315
12.1676 so_3887759_3887788_0 . . . 315
12.1677 so_15115859_15116012_0 . . . 315
12.1678 so_46649360_46649411_1 . . . 315
12.1679 so_4819936_4820339_5 . . . 315
12.1680 so_41991363_41991412_1 . . . 315
12.1681 so_26687426_26687426_0 . . . 315
12.1682 so_26687426_26687534_3 . . . 315
12.1683 so_33572861_33572962_2 . . . 315
12.1684 so_33572861_33572962_1 . . . 315
12.1685 so_5976459_5977975_5 . . . 315
12.1686 so_31838748_31839069_1 . . . 315
12.1687 so_28545644_28545644_0 . . . 315
12.1688 so_38864967_38865125_8 . . . 315
12.1689 so_18081114_18081262_1 . . . 315
12.1690 so_11309416_11309573_0 . . . 315
12.1691 so_14987847_14988094_1 . . . 315
12.1692 so_24237329_24237615_1 . . . 315
12.1693 so_31264984_31266254_0 . . . 315
12.1694 so_8849823_8849823_0 . . . 315
12.1695 so_28963577_28965673_6 . . . 315
12.1696 so_10761342_10761380_5 . . . 315

12.1697 so_19606492_19607079_3 . . . 315
12.1698 so_31864726_31865584_1 . . . 315
12.1699 so_23549400_23551597_0 . . . 315
12.1700 so_18895784_18895984_2 . . . 315
12.1701 so_21210360_21210565_1 . . . 315
12.1702 so_7297024_7297110_0 . . . 315
12.1703 so_28908636_28908765_0 . . . 315
12.1704 so_4338073_4338232_1 . . . 315
12.1705 so_4338073_4338232_0 . . . 315
12.1706 so_8797820_8798409_2 . . . 315
12.1707 so_3900192_3918423_7 . . . 315
12.1708 so_31866437_31866878_1 . . . 315
12.1709 so_26728495_26728495_3 . . . 315
12.1710 so_26728495_26728495_0 . . . 315
12.1711 so_7991666_7991919_5 . . . 315
12.1712 so_7991666_7991919_8 . . . 315
12.1713 so_36223198_36223967_0 . . . 315
12.1714 so_36223198_36223967_1 . . . 315
12.1715 so_27626949_27627173_2 . . . 315
12.1716 so_27626949_27627173_0 . . . 315
12.1717 so_3280768_3281119_5 . . . 315
12.1718 so_3280768_3281119_4 . . . 315
12.1719 so_10370713_10376279_1 . . . 315
12.1720 so_10370713_10376279_4 . . . 315
12.1721 so_46559313_46559427_3 . . . 315
12.1722 so_46559313_46559427_5 . . . 315
12.1723 so_46559313_46559427_4 . . . 315
12.1724 so_29868792_29873914_8 . . . 315
12.1725 so_29868792_29873914_9 . . . 315
12.1726 so_19411865_19413260_2 . . . 315
12.1727 so_6310638_6310703_0 . . . 315
12.1728 so_42647643_42647643_0 . . . 315
12.1729 so_24342941_24343032_5 . . . 315
12.1730 so_24342941_24343032_3 . . . 315
12.1731 so_24342941_24343032_1 . . . 315
12.1732 so_24342941_24343032_2 . . . 315
12.1733 so_24342941_24343032_4 . . . 315
12.1734 so_24342941_24343032_0 . . . 315
12.1735 so_29834274_29834456_0 . . . 315
12.1736 so_31571202_31571496_3 . . . 315
12.1737 so_21210310_21210394_0 . . . 315
12.1738 so_21210310_21210445_0 . . . 315
12.1739 so_13344800_13345578_0 . . . 315
12.1740 so_20608662_20609313_2 . . . 315
12.1741 so_20608662_20609313_0 . . . 315
12.1742 so_16703798_16703973_3 . . . 315
12.1743 so_16703798_16703973_2 . . . 315
12.1744 so_34746160_34749021_0 . . . 315
12.1745 so_13406987_13407454_1 . . . 315
12.1746 so_13406987_13407454_0 . . . 315
12.1747 so_25166969_25167071_0 . . . 315

12.1748 so_33957037_33957174_0 . . . 315
12.1749 so_41099950_41099950_0 . . . 315
12.1750 so_34810580_34811315_1 . . . 315
12.1751 so_3280768_3281119_6 . . . 315
12.1752 so_3280768_3281119_7 . . . 315
12.1753 so_4819936_4820339_0 . . . 315
12.1754 so_4819936_4819972_0 . . . 315
12.1755 so_4819936_4820339_2 . . . 315
12.1756 so_4819936_4820339_3 . . . 315
12.1757 so_4819936_4820339_1 . . . 315
12.1758 so_17478403_17478515_0 . . . 315
12.1759 so_25664694_25664886_3 . . . 315
12.1760 so_25664694_25664886_1 . . . 315
12.1761 so_25664694_25664886_0 . . . 315
12.1762 so_25664694_25664886_2 . . . 315
12.1763 so_25664694_25664886_4 . . . 315
12.1764 so_41991363_41991412_0 . . . 315
12.1765 so_5976459_5977975_2 . . . 315
12.1766 so_5976459_5977975_3 . . . 315
12.1767 so_5976459_5977975_1 . . . 315
12.1768 so_5976459_5977975_0 . . . 315
12.1769 so_5976459_5977975_6 . . . 315
12.1770 so_17221668_17222286_3 . . . 315
12.1771 so_38864967_38865125_6 . . . 315
12.1772 so_38864967_38865125_3 . . . 315
12.1773 so_38864967_38865125_5 . . . 315
12.1774 so_38864967_38865125_4 . . . 315
12.1775 so_25465703_25465763_0 . . . 315
12.1776 so_25465703_25465763_2 . . . 315
12.1777 so_25465703_25465763_1 . . . 315
12.1778 so_39417395_39417433_0 . . . 315
12.1779 so_33057562_33057644_3 . . . 315
12.1780 so_33057562_33057644_2 . . . 315
12.1781 so_33057562_33057646_0 . . . 315
12.1782 so_33057562_33057644_4 . . . 315
12.1783 so_33057562_33057644_0 . . . 315
12.1784 so_33057562_33057644_1 . . . 315
12.1785 so_42596060_42596389_0 . . . 315
12.1786 so_42596060_42596389_1 . . . 315
12.1787 so_38541354_38541830_4 . . . 315
12.1788 so_38541354_38541830_5 . . . 315
12.1789 so_38541354_38541830_6 . . . 315
12.1790 so_38541354_38541830_7 . . . 315
12.1791 so_11863435_11863938_3 . . . 315
12.1792 so_28963577_28965673_4 . . . 315
12.1793 so_28963577_28965673_2 . . . 315
12.1794 so_28963577_28965673_1 . . . 315
12.1795 so_28963577_28965673_3 . . . 315
12.1796 so_22111387_22111571_0 . . . 315
12.1797 so_37276413_37276413_0 . . . 315
12.1798 so_37276413_37276413_4 . . . 315

12.1799 so_37276413_37276413_2 . . . 315
12.1800 so_37276413_37276413_1 . . . 315
12.1801 so_37276413_37277533_1 . . . 315
12.1802 so_37276413_37277533_0 . . . 315
12.1803 so_37276413_37277174_0 . . . 315
12.1804 so_37276413_37276413_3 . . . 315
12.1805 so_27780647_27780739_0 . . . 315
12.1806 so_19606492_19607079_4 . . . 315
12.1807 so_35435255_35435597_1 . . . 315
12.1808 so_35435255_35435597_3 . . . 315
12.1809 so_31864726_31865584_0 . . . 315
12.1810 so_28908636_28908765_1 . . . 315
12.1811 so_33111800_33112112_0 . . . 315
12.1812 so_14085553_14085604_0 . . . 315
12.1813 so_45880152_45880590_1 . . . 315
12.1814 so_33063175_33063348_0 . . . 315
12.1815 so_27814280_27814376_0 . . . 315
12.1816 so_33539998_33539998_0 . . . 315
12.1817 so_41551545_41551678_4 . . . 316
12.1818 so_3280768_3281119_8 . . . 316
12.1819 so_24342941_24343032_9 . . . 316
12.1820 so_24342941_24343032_11 . . . 316
12.1821 so_24342941_24343032_7 . . . 316
12.1822 so_24342941_24343032_12 . . . 316
12.1823 so_24342941_24343032_6 . . . 316
12.1824 so_24342941_24343032_8 . . . 316
12.1825 so_13344800_13345578_1 . . . 316
12.1826 so_20608662_20609313_1 . . . 316
12.1827 so_20608662_20609313_4 . . . 316
12.1828 so_20608662_20609313_3 . . . 316
12.1829 so_23183322_23185451_1 . . . 316
12.1830 so_23183322_23185451_5 . . . 316
12.1831 so_23183322_23185451_2 . . . 316
12.1832 so_23183322_23185451_0 . . . 316
12.1833 so_23183322_23185451_3 . . . 316
12.1834 so_19559503_19559684_0 . . . 316
12.1835 so_13406987_13407454_2 . . . 316
12.1836 so_3280768_3281119_9 . . . 316
12.1837 so_4819936_4819972_1 . . . 316
12.1838 so_25803403_25803403_13 . . . 316
12.1839 so_24237329_24237615_5 . . . 316
12.1840 so_24237329_24237615_3 . . . 316
12.1841 so_24237329_24237615_7 . . . 316
12.1842 so_24237329_24237615_2 . . . 316
12.1843 so_24237329_24237615_4 . . . 316
12.1844 so_28555280_28555280_0 . . . 316
12.1845 so_28555280_28555280_2 . . . 316
12.1846 so_19040518_19041178_0 . . . 316
12.1847 so_46402983_46403511_3 . . . 316
12.1848 so_28476393_28477129_1 . . . 316
12.1849 so_28476393_28477129_2 . . . 316

12.1850 so_30252074_30252333_5 . . . 316
12.1851 so_30252074_30252333_1 . . . 316
12.1852 so_30252074_30252333_3 . . . 316
12.1853 so_30252074_30252333_2 . . . 316
12.1854 so_10402241_10402308_1 . . . 316
12.1855 so_10402241_10402308_3 . . . 316
12.1856 so_10402241_10402308_4 . . . 316
12.1857 so_10402241_10402308_2 . . . 316
12.1858 so_26879937_26879937_0 . . . 316
12.1859 so_26879937_26879937_2 . . . 316
12.1860 so_26879937_26881788_0 . . . 316
12.1861 so_32186400_32186876_1 . . . 316
12.1862 so_32186400_32186876_0 . . . 316
12.1863 so_19220311_19220387_2 . . . 316
12.1864 so_19220311_19220387_0 . . . 316
12.1865 so_23304025_23304090_4 . . . 316
12.1866 so_26942887_26942887_0 . . . 316
12.1867 so_21708606_21708606_0 . . . 316
12.1868 so_4338073_4338239_1 . . . 316
12.1869 so_26923320_26923320_0 . . . 316
12.1870 so_39139087_39139188_3 . . . 316
12.1871 so_39139087_39139214_0 . . . 316
12.1872 so_39139087_39139188_4 . . . 316
12.1873 so_31866437_31866437_0 . . . 316
12.1874 so_33539998_33539998_2 . . . 316
12.1875 so_33539998_33539998_1 . . . 316
12.1876 so_12754886_12755043_0 . . . 316
12.1877 so_12754886_12755118_0 . . . 316
12.1878 so_26994755_26995534_1 . . . 316
12.1879 so_26036384_26045084_1 . . . 316
12.1880 so_4178175_7189821_10 . . . 316
12.1881 so_4178175_7189821_13 . . . 316
12.1882 so_4178175_7189821_14 . . . 316
12.1883 so_4178175_7189821_12 . . . 316
12.1884 so_30096911_30097049_2 . . . 316
12.1885 so_16232547_16233624_3 . . . 316
12.1886 so_16232547_16233624_2 . . . 316
12.1887 so_16232547_16233624_1 . . . 316
12.1888 so_36098055_36098055_15 . . . 316
12.1889 so_36098055_36098055_11 . . . 316
12.1890 so_36098055_36098055_12 . . . 316
12.1891 so_36098055_36098055_9 . . . 316
12.1892 so_36098055_36098055_10 . . . 316
12.1893 so_36098055_36098055_7 . . . 316
12.1894 so_38043288_38084759_3 . . . 316
12.1895 so_29759441_29759556_2 . . . 316
12.1896 so_4275861_4417899_3 . . . 316
12.1897 so_4275861_4417899_2 . . . 316
12.1898 so_10402241_10402308_9 . . . 316
12.1899 so_10402241_10402308_8 . . . 316
12.1900 so_10402241_10402308_6 . . . 316

12.1901 so_30152687_30152747_2 . . . 316
12.1902 so_30152687_30152747_3 . . . 316
12.1903 so_22233148_22234327_4 . . . 316
12.1904 so_22233148_22234327_1 . . . 316
12.1905 so_16542291_16542385_8 . . . 316
12.1906 so_16542291_16542385_10 . . . 316
12.1907 so_16542291_16542385_11 . . . 316
12.1908 so_16542291_16542385_12 . . . 316
12.1909 so_31398388_31398779_10 . . . 316
12.1910 so_31398388_31398779_9 . . . 316
12.1911 so_37820806_37821190_0 . . . 316
12.1912 so_10854948_10855018_1 . . . 316
12.1913 so_24691820_24691907_1 . . . 316
12.1914 so_24691820_24691907_0 . . . 316
12.1915 so_18290523_25123890_0 . . . 316
12.1916 so_25465379_25465850_0 . . . 316
12.1917 so_37820806_37821190_1 . . . 316
12.1918 so_44863382_44863566_0 . . . 316
12.1919 so_15190486_15190614_1 . . . 316
12.1920 so_9167253_9167278_0 . . . 316
12.1921 so_18733227_18740867_0 . . . 316
12.1922 so_18733227_18740867_8 . . . 316
12.1923 so_18733227_18740867_1 . . . 316
12.1924 so_18733227_18740867_4 . . . 316
12.1925 so_18733227_18740867_5 . . . 316
12.1926 so_18733227_18740867_6 . . . 316
12.1927 so_18733227_18740867_7 . . . 316
12.1928 so_18733227_18740867_2 . . . 316
12.1929 so_18733227_18740867_3 . . . 316
12.1930 so_18733227_18740867_9 . . . 316
12.1931 so_34713108_34713427_3 . . . 316
12.1932 so_34713108_34713427_1 . . . 316
12.1933 so_34713108_34713427_2 . . . 316
12.1934 so_26299148_26299338_0 . . . 316
12.1935 so_4172722_4172724_1 . . . 316
12.1936 so_14754145_14754209_3 . . . 316
12.1937 so_37324393_37325450_0 . . . 316
12.1938 so_18290523_25123890_1 . . . 316
12.1939 so_18290523_25123890_2 . . . 316
12.1940 so_28457832_28458512_0 . . . 316
12.1941 so_19040518_19040518_1 . . . 316
12.1942 so_16055297_16055379_2 . . . 316
12.1943 so_16055297_16055393_1 . . . 316
12.1944 so_16055297_16055393_2 . . . 316
12.1945 so_16055297_16055393_0 . . . 316
12.1946 so_16055297_16055393_3 . . . 316
12.1947 so_16055297_16055393_4 . . . 316
12.1948 so_22111387_22111571_1 . . . 316
12.1949 so_26998571_26999748_1 . . . 316
12.1950 so_34993429_34995780_4 . . . 316
12.1951 so_34993429_34995780_3 . . . 316

12.1952 so_25324582_25325041_7 . . . 316
12.1953 so_46875027_46875219_0 . . . 316
12.1954 so_33379327_33379327_2 . . . 316
12.1955 so_33379327_34486446_0 . . . 316
12.1956 so_33379327_33379327_0 . . . 316
12.1957 so_33379327_33379327_3 . . . 316
12.1958 so_33379327_33379327_1 . . . 316
12.1959 so_18760657_18761525_1 . . . 316
12.1960 so_8218397_8218460_0 . . . 316
12.1961 so_1501920_1501970_0 . . . 316
12.1962 so_46875027_46875695_0 . . . 317
12.1963 so_11936730_11936784_2 . . . 317
12.1964 so_5625790_5625894_0 . . . 317
12.1965 so_24286523_24286824_0 . . . 317
12.1966 so_14369468_14369468_0 . . . 317
12.1967 so_47307601_47308482_1 . . . 317
12.1968 so_47273390_47274196_1 . . . 317
12.1969 so_10268523_10268562_0 . . . 317
12.1970 so_24789576_24789689_0 . . . 317
12.1971 so_9747406_9747527_0 . . . 317
12.1972 so_6090306_6090483_0 . . . 317
12.1973 so_29749580_29750200_0 . . . 317
12.1974 so_29749580_29749828_1 . . . 317
12.1975 so_7380450_7380475_1 . . . 317
12.1976 so_11172389_11172459_2 . . . 317
12.1977 so_7884102_7884102_0 . . . 317
12.1978 so_18283612_18283612_0 . . . 317
12.1979 so_20841770_20841815_1 . . . 317
12.1980 so_7080137_7080212_0 . . . 317
12.1981 so_3527248_3527705_1 . . . 317
12.1982 so_38541369_38541434_1 . . . 317
12.1983 so_26666406_26666439_1 . . . 317
12.1984 so_37507671_37507807_0 . . . 317
12.1985 so_37507671_37507825_0 . . . 317
12.1986 so_13267734_13267811_0 . . . 317
12.1987 so_33088586_33088753_0 . . . 317
12.1988 so_22186645_22186953_2 . . . 317
12.1989 so_23039658_23039770_0 . . . 317
12.1990 so_23039658_23039884_3 . . . 317
12.1991 so_8634134_8634157_1 . . . 317
12.1992 so_10136984_10137837_0 . . . 317
12.1993 so_19746474_19746828_1 . . . 317
12.1994 so_33539998_33542492_0 . . . 317
12.1995 so_26728495_26732327_0 . . . 317
12.1996 so_26728495_26732327_1 . . . 317
12.1997 so_26728495_26728495_2 . . . 317
12.1998 so_26728495_26728495_1 . . . 317
12.1999 so_704780_705611_5 . . . 317
12.2000 so_704780_705611_3 . . . 317
12.2001 so_11936730_11936784_3 . . . 317
12.2002 so_5625790_5625894_1 . . . 317

12.2003 so_24286523_24286824_1 . . . 317
12.2004 so_20502010_20502010_0 . . . 317
12.2005 so_27679002_27679053_0 . . . 317
12.2006 so_7991666_7991919_0 . . . 317
12.2007 so_7991666_7991832_0 . . . 317
12.2008 so_7991666_7991919_7 . . . 317
12.2009 so_41204025_41881445_4 . . . 317
12.2010 so_4178175_9491406_2 . . . 317
12.2011 so_32235294_32235317_2 . . . 317
12.2012 so_19267168_19267168_1 . . . 317
12.2013 so_24789576_24789689_1 . . . 317
12.2014 so_24789576_24789689_3 . . . 317
12.2015 so_9747406_9747527_2 . . . 317
12.2016 so_27665567_27665567_3 . . . 317
12.2017 so_27665567_27665567_1 . . . 317
12.2018 so_27665567_27665567_2 . . . 317
12.2019 so_7380450_7380491_0 . . . 317
12.2020 so_21013430_21013852_1 . . . 317
12.2021 so_21013430_21013943_1 . . . 317
12.2022 so_4124271_4124422_4 . . . 317
12.2023 so_3952020_3952020_0 . . . 317
12.2024 so_8867469_8867476_2 . . . 317
12.2025 so_8867469_8867476_0 . . . 317
12.2026 so_37970997_37971922_1 . . . 317
12.2027 so_2293796_2293926_0 . . . 317
12.2028 so_31838748_31839069_2 . . . 317
12.2029 so_10370713_10376279_3 . . . 317
12.2030 so_28545644_28545644_1 . . . 317
12.2031 so_18081114_18081262_2 . . . 317
12.2032 so_11309416_11309573_1 . . . 317
12.2033 so_26666406_26666439_0 . . . 317
12.2034 so_11863435_11863938_1 . . . 317
12.2035 so_33124542_33125217_0 . . . 317
12.2036 so_25606813_25606813_1 . . . 317
12.2037 so_19606492_19607079_5 . . . 317
12.2038 so_6310638_6310703_3 . . . 317
12.2039 so_6310638_6310703_5 . . . 317
12.2040 so_6310638_6310703_4 . . . 317
12.2041 so_19590008_19590429_1 . . . 317
12.2042 so_4338073_4338547_0 . . . 317
12.2043 so_17220498_17220579_7 . . . 317
12.2044 so_17220498_17220579_1 . . . 317
12.2045 so_17220498_17220579_4 . . . 317
12.2046 so_17220498_17220579_6 . . . 317
12.2047 so_31866437_31866878_2 . . . 317
12.2048 so_9747406_9747527_1 . . . 318
12.2049 so_38541369_38541434_3 . . . 318
12.2050 so_37507671_37507807_1 . . . 318
12.2051 so_5625790_5625834_1 . . . 318
12.2052 so_44863382_44863678_3 . . . 318
12.2053 so_44863382_44863678_2 . . . 318

12.2054 so_25395805_25395805_2 . . . 318
12.2055 so_9747406_9747527_3 . . . 318
12.2056 so_25166969_25167071_1 . . . 318
12.2057 so_3280768_3281119_3 . . . 318
12.2058 so_17478403_17478515_1 . . . 318
12.2059 so_2953611_2953925_3 . . . 318
12.2060 so_38541354_38541830_3 . . . 318
12.2061 so_11863435_11863938_2 . . . 318
12.2062 so_27780647_27780739_1 . . . 318
12.2063 so_19606492_19607079_6 . . . 318
12.2064 so_28908636_28908765_2 . . . 318
12.2065 so_4338073_4338239_0 . . . 318
12.2066 so_33063175_33063175_3 . . . 318
12.2067 so_27792604_27792948_1 . . . 318
12.2068 so_4124271_4124422_3 . . . 318
12.2069 so_4124271_4124422_1 . . . 318
12.2070 so_15022060_15022107_1 . . . 318
12.2071 so_37970997_37971922_2 . . . 318
12.2072 so_10781130_10781145_0 . . . 318
12.2073 so_46402983_46403511_4 . . . 318
12.2074 so_26666406_26666439_2 . . . 318
12.2075 so_14566341_14566341_2 . . . 318
12.2076 so_25606813_25606873_0 . . . 318
12.2077 so_31732576_31732818_0 . . . 318
12.2078 so_8894090_26375167_0 . . . 319
12.2079 so_24789576_24789689_4 . . . 319
12.2080 so_394854_394867_0 . . . 319
12.2081 so_30096911_30097049_3 . . . 319
12.2082 so_28334899_28336094_1 . . . 319
12.2083 so_36098055_36098055_16 . . . 319
12.2084 so_36098055_36098055_18 . . . 319
12.2085 so_36098055_36098055_13 . . . 319
12.2086 so_36098055_36098055_17 . . . 319
12.2087 so_29759441_29759556_3 . . . 319
12.2088 so_30152687_30152747_5 . . . 319
12.2089 so_30152687_30152747_4 . . . 319
12.2090 so_31398388_31398779_11 . . . 319
12.2091 so_31398388_31398779_12 . . . 319
12.2092 so_15190486_15190614_2 . . . 319
12.2093 so_9167253_9167278_1 . . . 319
12.2094 so_3437110_3437197_2 . . . 319
12.2095 so_28091889_28092512_1 . . . 319
12.2096 so_24789576_24789689_5 . . . 319
12.2097 so_26299148_26299338_1 . . . 319
12.2098 so_4172722_4172724_2 . . . 319
12.2099 so_14754145_14754209_4 . . . 319
12.2100 so_30194370_30195104_1 . . . 319
12.2101 so_30194370_30195104_0 . . . 319
12.2102 so_34411588_34412317_1 . . . 319
12.2103 so_36095290_36095358_2 . . . 319
12.2104 so_8867469_8867476_3 . . . 319

12.2105 so_8867469_8867476_4 . . . 319
12.2106 so_10781130_10781727_1 . . . 319
12.2107 so_10781130_10781727_3 . . . 319
12.2108 so_10781130_10781727_2 . . . 319
12.2109 so_10781130_10781727_4 . . . 319
12.2110 so_44394079_44394682_0 . . . 319
12.2111 so_13559255_13559392_2 . . . 319
12.2112 so_13559255_13559392_0 . . . 319
12.2113 so_13559255_13559392_1 . . . 319
12.2114 so_13559255_13559392_3 . . . 319
12.2115 so_16957812_16958411_0 . . . 319
12.2116 so_34491686_34491733_2 . . . 319
12.2117 so_34491686_34491733_1 . . . 319
12.2118 so_42575122_42575122_1 . . . 319
12.2119 so_18760657_18761525_0 . . . 319
12.2120 so_18079769_18080046_2 . . . 319
12.2121 so_18079769_18080046_0 . . . 319
12.2122 so_33936295_33937081_2 . . . 320
12.2123 so_2671532_2671694_0 . . . 320
12.2124 so_4639576_4639839_0 . . . 320
12.2125 so_16089730_16112511_0 . . . 320
12.2126 so_10292646_10292865_0 . . . 320
12.2127 so_6163040_6163385_0 . . . 320
12.2128 so_29747503_29747759_0 . . . 320
12.2129 so_19267408_19272035_0 . . . 320
12.2130 so_42652965_42653466_5 . . . 320
12.2131 so_42652965_42653466_3 . . . 320
12.2132 so_33873047_33879489_0 . . . 320
12.2133 so_12011426_12011561_0 . . . 320
12.2134 so_33371684_33371684_0 . . . 320
12.2135 so_11028929_11029028_2 . . . 320
12.2136 so_13567029_13567029_0 . . . 320
12.2137 so_15718262_15718279_9 . . . 320
12.2138 so_13898750_13898867_0 . . . 320
12.2139 so_45807900_45828592_1 . . . 320
12.2140 so_10201321_10201402_0 . . . 320
12.2141 so_12953127_12953145_6 . . . 320
12.2142 so_12953127_12953145_5 . . . 320
12.2143 so_12953127_12953145_2 . . . 320
12.2144 so_12953127_12953145_3 . . . 320
12.2145 so_12953127_12953145_0 . . . 320
12.2146 so_12953127_12953145_4 . . . 320
12.2147 so_12953127_12953145_1 . . . 320
12.2148 so_881966_882055_0 . . . 320
12.2149 so_881966_882055_1 . . . 320
12.2150 so_881966_882055_2 . . . 320
12.2151 so_2145030_2145460_0 . . . 320
12.2152 so_22193569_22193569_0 . . . 320
12.2153 so_27045136_27046690_3 . . . 320
12.2154 so_27045136_27046690_0 . . . 320
12.2155 so_27045136_27046690_1 . . . 320

12.2156 so_35852493_35853459_2 . . . 320
12.2157 so_8760813_8760813_0 . . . 320
12.2158 so_8760813_8760854_2 . . . 320
12.2159 so_47529358_47529358_6 . . . 320
12.2160 so_47529358_47529358_0 . . . 320
12.2161 so_47529358_47529358_3 . . . 320
12.2162 so_47529358_47529358_5 . . . 320
12.2163 so_47529358_47529358_2 . . . 320
12.2164 so_37185072_37185298_2 . . . 320
12.2165 so_7401521_7401764_0 . . . 320
12.2166 so_7401521_7401782_1 . . . 320
12.2167 so_19445070_19445448_5 . . . 320
12.2168 so_19445070_19445448_4 . . . 320
12.2169 so_12686099_12686210_1 . . . 320
12.2170 so_32364011_32364220_0 . . . 320
12.2171 so_23335589_23336083_0 . . . 320
12.2172 so_23335589_23337348_0 . . . 320
12.2173 so_25829479_25830595_1 . . . 320
12.2174 so_25829479_25830595_0 . . . 320
12.2175 so_23475776_23476013_0 . . . 320
12.2176 so_24722132_24722282_0 . . . 320
12.2177 so_46628553_46628630_2 . . . 320
12.2178 so_33891225_33891582_2 . . . 320
12.2179 so_33891225_33891582_1 . . . 320
12.2180 so_26959846_26959914_0 . . . 320
12.2181 so_28651855_28652030_1 . . . 320
12.2182 so_28651855_28652030_0 . . . 320
12.2183 so_16234323_16235654_0 . . . 320
12.2184 so_16234323_16235654_1 . . . 320
12.2185 so_27249698_27249953_0 . . . 320
12.2186 so_15040784_15040860_0 . . . 320
12.2187 so_11492948_11493036_0 . . . 320
12.2188 so_47508477_47510004_0 . . . 320
12.2189 so_4316727_4316948_1 . . . 320
12.2190 so_15412762_15412927_1 . . . 320
12.2191 so_28412305_28412305_0 . . . 320
12.2192 so_28412305_28412305_1 . . . 320
12.2193 so_27963731_27963942_0 . . . 320
12.2194 so_27963731_27963942_2 . . . 320
12.2195 so_21206359_21206750_2 . . . 320
12.2196 so_2671532_2671694_1 . . . 320
12.2197 so_20043129_20043180_0 . . . 320
12.2198 so_7140607_7140784_0 . . . 320
12.2199 so_15098897_15098927_0 . . . 320
12.2200 so_27195825_27206105_2 . . . 320
12.2201 so_27195825_27206105_0 . . . 320
12.2202 so_24925137_24925451_2 . . . 320
12.2203 so_6352438_6359094_0 . . . 320
12.2204 so_12356332_12356332_1 . . . 320
12.2205 so_13099603_13099664_0 . . . 320
12.2206 so_13099603_13099752_0 . . . 320

12.2207 so_2722879_30948607_4 . . . 320
12.2208 so_24172205_24172639_2 . . . 320
12.2209 so_24172205_24172639_0 . . . 320
12.2210 so_24172205_24172519_0 . . . 320
12.2211 so_24172205_24172639_1 . . . 320
12.2212 so_12288131_12288328_1 . . . 320
12.2213 so_29562908_29563111_2 . . . 320
12.2214 so_29562908_29563111_1 . . . 320
12.2215 so_29562908_29563111_0 . . . 320
12.2216 so_9578047_9578101_0 . . . 320
12.2217 so_31520406_31520617_1 . . . 320
12.2218 so_25534628_25534823_6 . . . 320
12.2219 so_26193117_26193285_0 . . . 320
12.2220 so_20455382_20455655_2 . . . 320
12.2221 so_20455382_20455496_0 . . . 320
12.2222 so_20455382_20455655_1 . . . 320
12.2223 so_24043577_24043750_4 . . . 320
12.2224 so_11914691_11915906_1 . . . 320
12.2225 so_3905869_3906038_2 . . . 320
12.2226 so_3905869_3906038_3 . . . 320
12.2227 so_3905869_3906038_1 . . . 320
12.2228 so_6009004_6009012_0 . . . 320
12.2229 so_19338510_19338860_0 . . . 320
12.2230 so_19338510_19338860_1 . . . 320
12.2231 so_26324347_26324405_0 . . . 320
12.2232 so_9563600_9563914_6 . . . 320
12.2233 so_31056036_31056162_0 . . . 320
12.2234 so_33344259_33344927_0 . . . 320
12.2235 so_11791644_11791712_0 . . . 320
12.2236 so_32759433_32761541_1 . . . 320
12.2237 so_6139188_6139228_0 . . . 320
12.2238 so_23249124_23249559_0 . . . 320
12.2239 so_7457370_7457436_4 . . . 320
12.2240 so_6383639_6383710_2 . . . 320
12.2241 so_6383639_6383710_1 . . . 320
12.2242 so_6383639_6383710_0 . . . 320
12.2243 so_40907276_40908958_1 . . . 320
12.2244 so_19169421_19169458_0 . . . 320
12.2245 so_30380631_30380688_3 . . . 320
12.2246 so_8924034_8924348_1 . . . 320
12.2247 so_8924034_8924348_4 . . . 320
12.2248 so_8924034_8924348_3 . . . 320
12.2249 so_8924034_8924348_2 . . . 320
12.2250 so_8924034_8924348_0 . . . 320
12.2251 so_44475982_44476033_0 . . . 320
12.2252 so_42727765_42728073_0 . . . 320
12.2253 so_37871609_38040678_6 . . . 320
12.2254 so_12881210_25321049_0 . . . 320
12.2255 so_33086346_33086461_0 . . . 320
12.2256 so_33872026_33872152_0 . . . 320
12.2257 so_32812049_32812491_0 . . . 320

12.2258 so_20484153_20484272_0 . . . 320
12.2259 so_40784322_40785248_1 . . . 320
12.2260 so_40784322_40784990_0 . . . 320
12.2261 so_43163801_43163920_1 . . . 320
12.2262 so_38801955_38805510_0 . . . 320
12.2263 so_3605731_3606185_0 . . . 320
12.2264 so_21907836_21908272_1 . . . 320
12.2265 so_21907836_21908272_0 . . . 320
12.2266 so_46683353_46683875_0 . . . 320
12.2267 so_19792135_19792135_0 . . . 320
12.2268 so_16938951_16939177_1 . . . 320
12.2269 so_28539763_28539763_0 . . . 320
12.2270 so_4809120_4809146_1 . . . 320
12.2271 so_34969278_34969407_0 . . . 320
12.2272 so_41817052_41817619_0 . . . 320
12.2273 so_44922222_44922513_0 . . . 320
12.2274 so_44689565_44689617_0 . . . 320
12.2275 so_14012309_14012334_1 . . . 320
12.2276 so_26937270_26937418_0 . . . 320
12.2277 so_28875810_28876101_1 . . . 320
12.2278 so_15004242_15004366_1 . . . 320
12.2279 so_15004242_15004366_0 . . . 320
12.2280 so_11016262_11016416_0 . . . 320
12.2281 so_32185160_32185404_0 . . . 320
12.2282 so_18486598_18486996_9 . . . 320
12.2283 so_18486598_18487088_0 . . . 320
12.2284 so_18486598_18486996_8 . . . 320
12.2285 so_18486598_18486996_6 . . . 320
12.2286 so_18486598_18486996_1 . . . 320
12.2287 so_18486598_18486996_7 . . . 320
12.2288 so_18486598_18486996_4 . . . 320
12.2289 so_18486598_18486996_2 . . . 320
12.2290 so_18486598_18486996_0 . . . 320
12.2291 so_18486598_18486996_3 . . . 320
12.2292 so_35744832_35745378_5 . . . 320
12.2293 so_11262367_11262385_0 . . . 320
12.2294 so_5770634_5770634_0 . . . 320
12.2295 so_42066811_42067082_5 . . . 320
12.2296 so_42066811_42067082_8 . . . 320
12.2297 so_42066811_42067082_7 . . . 320
12.2298 so_42114278_42114332_0 . . . 320
12.2299 so_42114278_42114325_0 . . . 320
12.2300 so_26493902_26494575_0 . . . 320
12.2301 so_10292646_10292858_1 . . . 320
12.2302 so_10292646_10292859_0 . . . 320
12.2303 so_10292646_10292646_0 . . . 320
12.2304 so_10292646_10292859_2 . . . 320
12.2305 so_10292646_10292859_1 . . . 320
12.2306 so_14154290_14154318_0 . . . 320
12.2307 so_18849958_18850149_1 . . . 320
12.2308 so_18849958_18850149_0 . . . 320

12.2309 so_18849958_18850149_2 . . . 320
12.2310 so_33965966_33966029_1 . . . 320
12.2311 so_33965966_33966029_0 . . . 320
12.2312 so_32370434_32370752_1 . . . 320
12.2313 so_27695528_27698437_1 . . . 320
12.2314 so_27695528_27698437_2 . . . 320
12.2315 so_36699989_36700919_0 . . . 320
12.2316 so_27633297_27633297_1 . . . 320
12.2317 so_1758142_9945598_1 . . . 320
12.2318 so_20874193_20874207_0 . . . 320
12.2319 so_20649951_20650181_0 . . . 320
12.2320 so_41519368_41520375_0 . . . 320
12.2321 so_185624_189162_10 . . . 320
12.2322 so_11509757_11509757_0 . . . 320
12.2323 so_17677730_17678970_5 . . . 320
12.2324 so_17677730_17678970_4 . . . 320
12.2325 so_17677730_17678970_6 . . . 320
12.2326 so_12011426_12011561_2 . . . 321
12.2327 so_14912800_14913148_3 . . . 321
12.2328 so_11028929_11029028_3 . . . 321
12.2329 so_45807900_45828592_0 . . . 321
12.2330 so_9532608_9532647_0 . . . 321
12.2331 so_29717144_29717851_0 . . . 321
12.2332 so_25875596_25876175_1 . . . 321
12.2333 so_25875596_25876175_0 . . . 321
12.2334 so_9152798_9152883_0 . . . 321
12.2335 so_46107480_46107646_0 . . . 321
12.2336 so_19445070_19445448_2 . . . 321
12.2337 so_19445070_19445448_3 . . . 321
12.2338 so_32658772_32659136_0 . . . 321
12.2339 so_46628553_46628630_3 . . . 321
12.2340 so_46628553_46628630_1 . . . 321
12.2341 so_32844948_32845247_0 . . . 321
12.2342 so_11492948_11493012_0 . . . 321
12.2343 so_4316727_4316948_2 . . . 321
12.2344 so_28412305_28412305_2 . . . 321
12.2345 so_20043129_20043180_1 . . . 321
12.2346 so_15098897_15098927_1 . . . 321
12.2347 so_36752678_36752874_0 . . . 321
12.2348 so_6352438_6359094_2 . . . 321
12.2349 so_35040376_35040690_0 . . . 321
12.2350 so_15591603_15591741_0 . . . 321
12.2351 so_14967167_14967452_4 . . . 321
12.2352 so_12288131_12288328_0 . . . 321
12.2353 so_25534628_25534823_5 . . . 321
12.2354 so_37932066_37932254_0 . . . 321
12.2355 so_22188643_22189404_1 . . . 321
12.2356 so_11914691_11914691_0 . . . 321
12.2357 so_6009004_6009012_1 . . . 321
12.2358 so_20019791_20020230_0 . . . 321
12.2359 so_5266783_5267706_1 . . . 321

12.2360 so_46071992_46071992_0 . . . 321
12.2361 so_32759433_32761541_2 . . . 321
12.2362 so_25885065_25885212_0 . . . 321
12.2363 so_23914289_23915503_1 . . . 321
12.2364 so_35906268_35906884_1 . . . 321
12.2365 so_35906268_35906884_0 . . . 321
12.2366 so_11817873_19698477_1 . . . 321
12.2367 so_31546504_31546666_0 . . . 321
12.2368 so_39800666_39801353_0 . . . 321
12.2369 so_26733576_26734444_0 . . . 321
12.2370 so_17481018_17483612_0 . . . 321
12.2371 so_28539763_28539763_1 . . . 321
12.2372 so_28718507_28719283_0 . . . 321
12.2373 so_40039379_40041585_0 . . . 321
12.2374 so_40039379_40039379_0 . . . 321
12.2375 so_14012309_14012334_2 . . . 321
12.2376 so_46097854_46099015_2 . . . 321
12.2377 so_33971318_33972072_0 . . . 321
12.2378 so_15004242_15004366_2 . . . 321
12.2379 so_24207962_24208281_0 . . . 321
12.2380 so_29164188_29165754_0 . . . 321
12.2381 so_29834308_29834426_0 . . . 321
12.2382 so_41283712_41284800_1 . . . 321
12.2383 so_41283712_41302628_0 . . . 321
12.2384 so_41283712_41284800_3 . . . 321
12.2385 so_26493902_26494575_1 . . . 321
12.2386 so_38617309_38617484_0 . . . 321
12.2387 so_11088023_11088307_0 . . . 321
12.2388 so_11088023_11088507_0 . . . 321
12.2389 so_7935639_7935954_0 . . . 321
12.2390 so_28410567_28410620_2 . . . 321
12.2391 so_27695528_27698437_0 . . . 321
12.2392 so_11247654_11247831_1 . . . 321
12.2393 so_9183087_9183087_0 . . . 321
12.2394 so_27633297_27633297_0 . . . 321
12.2395 so_31775876_31775876_0 . . . 321
12.2396 so_41519368_41523082_3 . . . 321
12.2397 so_41519368_41523082_1 . . . 321
12.2398 so_11509757_11509757_1 . . . 321
12.2399 so_45179514_45179751_2 . . . 321
12.2400 Inheriting constructors . . . 322
12.2401 so_20062515_20062581_0 . . . 322
12.2402 so_30554119_30554202_0 . . . 322
12.2403 so_31830780_31831724_0 . . . 322
12.2404 so_24165504_24165617_6 . . . 322
12.2405 so_24165504_24165617_4 . . . 322
12.2406 so_24165504_24165617_2 . . . 322
12.2407 so_32894131_32894445_2 . . . 322
12.2408 so_41495401_41495871_1 . . . 322
12.2409 so_40932844_40933118_0 . . . 322
12.2410 so_24912280_25657097_0 . . . 322

12.241 so_18313509_18313509_0 . . . 322
12.242 so_24635935_24636309_0 . . . 322
12.243 so_16447145_16447696_0 . . . 322
12.244 so_16780786_16781321_1 . . . 322
12.245 so_20255173_20255756_0 . . . 322
12.246 so_44522331_44522662_0 . . . 322
12.247 so_37585945_37586303_0 . . . 322
12.248 so_42029637_42030177_0 . . . 322
12.249 so_22534997_22536531_2 . . . 322
12.250 so_22534997_22536531_1 . . . 322
12.251 so_20435859_20436119_0 . . . 322
12.252 so_24333458_24333884_1 . . . 322
12.253 so_41495401_41495871_5 . . . 322
12.254 so_33376021_33376246_0 . . . 322
12.255 so_14963327_14963372_1 . . . 322
12.256 so_42791818_42792049_0 . . . 322
12.257 so_28450803_28450893_0 . . . 322
12.258 so_22110903_22110962_0 . . . 322
12.259 so_29477548_29478161_0 . . . 322
12.260 so_24014240_24015931_0 . . . 322
12.261 so_42869338_42869338_0 . . . 322
12.262 so_24635935_24636309_1 . . . 322
12.263 so_24635935_24636348_0 . . . 322
12.264 so_33376021_33376246_1 . . . 322
12.265 so_24333458_24333884_2 . . . 322
12.266 so_20255173_20255756_2 . . . 322
12.267 so_20255173_20255756_1 . . . 322
12.268 so_24912280_25657097_1 . . . 324
12.269 so_27665567_27678581_0 . . . 324
12.270 so_27003791_27003791_0 . . . 324
12.271 so_25653647_25653843_1 . . . 324
12.272 so_40932844_40933634_0 . . . 324
12.273 so_27957866_27958350_0 . . . 324
12.274 so_37064993_37065579_0 . . . 324
12.275 so_7394492_7394577_0 . . . 324

13 Overloading **326**
13.1 so_31343142_31348802_3 . . . 326
13.2 so_26009080_26009080_0 . . . 326
13.3 so_5406533_5911476_2 . . . 326
13.4 so_35373384_35375162_5 . . . 326
13.5 so_37846565_37846679_0 . . . 326
13.6 so_43793941_43794166_1 . . . 326
13.7 so_23973281_23976676_0 . . . 326
13.8 so_23966271_23966948_0 . . . 326
13.9 so_13410460_13410529_0 . . . 326
13.10 so_38035363_38035431_1 . . . 326
13.11 so_35373384_35375162_6 . . . 326
13.12 Overloadable declarations . . . 326
13.13 so_31343142_31348802_4 . . . 326
13.14 so_31453992_31454259_0 . . . 326

13.15 so_27728534_27728534_0 . . . 326
13.16 so_31062892_31062984_0 . . . 326
13.17 so_1429547_1429617_0 . . . 326
13.18 so_32776445_32776445_0 . . . 326
13.19 so_19464116_19464360_0 . . . 326
13.20 so_27781485_27781584_1 . . . 326
13.21 so_23066901_23066981_0 . . . 326
13.22 so_23066901_23066981_2 . . . 326
13.23 so_23066901_23066981_1 . . . 326
13.24 so_22446891_22447103_0 . . . 326
13.25 so_44478304_44478424_0 . . . 326
13.26 so_40762477_40762531_2 . . . 326
13.27 so_40762477_40762531_0 . . . 326
13.28 so_40762477_40762531_1 . . . 326
13.29 so_37038832_37038832_0 . . . 326
13.30 so_20873870_20873926_0 . . . 326
13.31 so_4331837_4332342_0 . . . 326
13.32 so_5365689_5365714_1 . . . 326
13.33 so_18432894_18432953_0 . . . 326
13.34 so_27716871_27716871_0 . . . 326
13.35 so_31062892_31062984_1 . . . 326
13.36 so_36479812_36479812_0 . . . 326
13.37 so_282800_26695750_0 . . . 326
13.38 so_1429547_1429617_1 . . . 327
13.39 so_10242578_10242660_2 . . . 327
13.40 so_10242578_10242660_0 . . . 327
13.41 so_10242578_10242660_3 . . . 327
13.42 so_33648117_33661933_0 . . . 327
13.43 so_9327505_9328713_0 . . . 327
13.44 so_9327505_9327552_0 . . . 327
13.45 so_19464116_19464360_2 . . . 327
13.46 so_19464116_19464360_1 . . . 327
13.47 so_23840421_23840460_0 . . . 327
13.48 so_7321993_7322098_0 . . . 327
13.49 so_16821956_16821998_0 . . . 327
13.50 so_15862549_15862607_1 . . . 327
13.51 so_37913596_37914489_1 . . . 327
13.52 so_37913596_37914489_0 . . . 327
13.53 so_30352301_30352621_1 . . . 327
13.54 so_28491974_28492090_2 . . . 327
13.55 so_28491974_28492090_0 . . . 327
13.56 so_20659000_20659598_1 . . . 327
13.57 so_20659000_20659598_0 . . . 327
13.58 so_24688557_24688652_2 . . . 327
13.59 so_27398049_27398497_2 . . . 327
13.60 so_27398049_27398497_4 . . . 327
13.61 so_10582440_10582759_0 . . . 327
13.62 so_2627166_3598165_1 . . . 327
13.63 so_24765077_24765174_1 . . . 327
13.64 so_24765077_24765174_0 . . . 327
13.65 so_14987847_14988094_0 . . . 327

13.66 so_34960981_34961245_2 . . . 327
13.67 so_38927019_38930181_0 . . . 327
13.68 so_1353407_8108285_0 . . . 327
13.69 so_8143711_8143817_0 . . . 327
13.70 so_27773915_27773915_0 . . . 327
13.71 so_706059_706070_0 . . . 327
13.72 so_706059_706070_1 . . . 327
13.73 so_706059_706100_0 . . . 327
13.74 so_1790704_1790908_0 . . . 327
13.75 so_7459269_7459310_0 . . . 327
13.76 so_11645751_11645914_1 . . . 327
13.77 so_11645751_11645914_2 . . . 327
13.78 so_11645751_11645914_0 . . . 327
13.79 so_7557968_7558010_0 . . . 327
13.80 so_11535388_11535695_1 . . . 327
13.81 so_28655541_28658366_2 . . . 327
13.82 so_3682049_3682076_2 . . . 327
13.83 so_3682049_3682076_1 . . . 327
13.84 so_3682049_3682076_0 . . . 327
13.85 Declaration matching . . . 329
13.86 so_14150134_14150169_0 . . . 329
13.87 so_35373384_35375162_9 . . . 329
13.88 Overload resolution . . . 330
13.89 so_6239630_6239654_0 . . . 330
13.90 so_10854354_10854430_0 . . . 330
13.91 so_39042240_39042976_1 . . . 330
13.92 so_16922121_16922652_0 . . . 330
13.93 so_42573771_42574950_0 . . . 330
13.94 so_35592392_35592630_0 . . . 330
13.95 so_2169932_2170050_0 . . . 330
13.96 so_25552822_25553119_4 . . . 330
13.97 so_25552822_25553119_3 . . . 330
13.98 so_25552822_25553119_2 . . . 330
13.99 so_18089988_18090865_6 . . . 330
13.100 so_18089988_18090865_5 . . . 330
13.101 so_18089988_18090865_7 . . . 330
13.102 so_18089988_18090865_2 . . . 330
13.103 so_25299888_25299996_3 . . . 330
13.104 so_11296389_12610403_1 . . . 330
13.105 so_11296389_12610403_2 . . . 330
13.106 so_29522800_29523073_2 . . . 330
13.107 so_29522800_29523073_6 . . . 330
13.108 so_29522800_29523073_4 . . . 330
13.109 so_29522800_29523073_7 . . . 330
13.110 so_29522800_29523073_1 . . . 330
13.111 so_29522800_29523073_8 . . . 330
13.112 so_29522800_29523073_3 . . . 330
13.113 so_29522800_29523073_5 . . . 330
13.114 so_13900616_13900722_0 . . . 330
13.115 so_33242948_33243400_3 . . . 330
13.116 so_33242948_33243400_4 . . . 330

13.117 so_33242948_33243400_1 . . . 330
13.118 so_33242948_33243400_2 . . . 330
13.119 so_33242948_33243400_5 . . . 330
13.120 so_16088636_16088846_1 . . . 330
13.121 so_16088636_16088846_0 . . . 330
13.122 so_644461_644476_0 . . . 330
13.123 so_46409188_46409188_0 . . . 330
13.124 so_30829648_30830594_0 . . . 330
13.125 so_28008073_28008073_0 . . . 330
13.126 so_12968135_12968346_0 . . . 330
13.127 so_34147715_34147799_1 . . . 330
13.128 so_41594186_41707564_3 . . . 330
13.129 so_17878750_17879145_0 . . . 330
13.130 so_31343142_31348802_7 . . . 330
13.131 so_39042240_39042976_0 . . . 330
13.132 so_644461_644476_1 . . . 330
13.133 so_25316779_25316835_0 . . . 330
13.134 so_46054228_46054902_0 . . . 330
13.135 so_47002799_47003980_1 . . . 330
13.136 so_18759874_18759989_1 . . . 331
13.137 so_46579016_46580007_1 . . . 331
13.138 so_15028835_15029228_1 . . . 331
13.139 so_8914986_8917084_1 . . . 331
13.140 so_19955954_19958120_0 . . . 331
13.141 so_27972910_27973022_0 . . . 331
13.142 so_25110429_25110733_1 . . . 331
13.143 so_19433014_19434526_0 . . . 331
13.144 so_27773915_27773915_1 . . . 331
13.145 so_8610571_8610728_0 . . . 331
13.146 so_19276238_19276546_0 . . . 331
13.147 so_25241516_25242107_9 . . . 331
13.148 so_46579016_46580007_2 . . . 331
13.149 so_8914986_8917084_2 . . . 331
13.150 so_11911279_11911331_0 . . . 331
13.151 so_8610571_8610728_1 . . . 331
13.152 so_19276238_19276546_1 . . . 331
13.153 so_5683687_5683687_0 . . . 331
13.154 so_30393443_30394180_1 . . . 331
13.155 so_30393443_30394180_0 . . . 331
13.156 so_39349005_39349199_0 . . . 331
13.157 so_21861148_21861190_2 . . . 331
13.158 so_21861148_21861190_1 . . . 331
13.159 so_46579016_46580007_3 . . . 331
13.160 so_46579016_46580007_4 . . . 331
13.161 so_21469135_21469596_0 . . . 331
13.162 so_19955954_19958120_1 . . . 331
13.163 so_19955954_19958120_4 . . . 331
13.164 so_19955954_19958120_3 . . . 331
13.165 so_19955954_19958120_2 . . . 331
13.166 so_17130607_17130847_3 . . . 331
13.167 so_17130607_17130847_1 . . . 331

13.168 so_17130607_17130847_0 . . . 331
13.169 so_17130607_17130847_2 . . . 331
13.170 so_17147069_17147069_0 . . . 331
13.171 so_27972910_27973022_3 . . . 331
13.172 so_27972910_27973022_1 . . . 331
13.173 so_27972910_27973022_2 . . . 331
13.174 so_27972910_27973022_4 . . . 331
13.175 so_27678257_27678304_0 . . . 331
13.176 so_35389911_35404544_1 . . . 331
13.177 so_35389911_35403879_2 . . . 331
13.178 so_35389911_35403879_3 . . . 331
13.179 so_35389911_35404544_2 . . . 331
13.180 so_35389911_35403879_1 . . . 331
13.181 so_35389911_35404544_0 . . . 331
13.182 so_30093923_30094209_1 . . . 331
13.183 so_30093923_30094209_0 . . . 331
13.184 so_12399372_12399395_0 . . . 331
13.185 so_32699007_32699790_2 . . . 331
13.186 so_3544824_3544824_1 . . . 331
13.187 so_47002799_47003980_3 . . . 331
13.188 so_47002799_47003980_5 . . . 331
13.189 so_47002799_47003980_4 . . . 331
13.190 so_47002799_47003980_2 . . . 331
13.191 so_11911279_11911331_3 . . . 331
13.192 so_11911279_11911331_1 . . . 331
13.193 so_11911279_11911331_2 . . . 331
13.194 so_8610571_8610728_5 . . . 331
13.195 so_8610571_8610728_4 . . . 331
13.196 so_8610571_8610728_3 . . . 331
13.197 so_8610571_8610728_2 . . . 331
13.198 so_26923320_26923549_2 . . . 331
13.199 so_25241516_25242107_11 . . . 331
13.200 so_25241516_25242107_10 . . . 331
13.201 so_25241516_25242107_12 . . . 331
13.202 so_27317838_27317838_0 . . . 331
13.203 so_30393443_30394180_3 . . . 331
13.204 so_30393443_30394180_2 . . . 331
13.205 so_27972910_27973022_5 . . . 331
13.206 so_27678257_27678304_2 . . . 331
13.207 so_35389911_35404544_3 . . . 331
13.208 so_35389911_35404544_4 . . . 331
13.209 so_7157691_7158915_0 . . . 331
13.210 so_47002799_47003980_6 . . . 331
13.211 so_8610571_8610728_6 . . . 331
13.212 so_8610571_8610728_10 . . . 331
13.213 so_8610571_8610728_8 . . . 331
13.214 so_8610571_8610728_7 . . . 331
13.215 so_28259765_28259765_1 . . . 331
13.216 so_28259765_28259765_0 . . . 331
13.217 so_28259765_28259765_2 . . . 331
13.218 so_19276238_19276546_3 . . . 331

13.219 so_19276238_19276546_4 . . . 331
13.220 so_19276238_19276546_2 . . . 331
13.221 so_25241516_25242022_2 . . . 331
13.222 so_9975291_9975452_2 . . . 332
13.223 so_25836109_25836942_4 . . . 332
13.224 so_39150553_39276652_2 . . . 332
13.225 so_3723630_3723905_0 . . . 332
13.226 so_28079317_28086248_0 . . . 332
13.227 so_25110429_25110733_3 . . . 332
13.228 so_25110429_25110733_2 . . . 332
13.229 so_14971536_14973116_0 . . . 332
13.230 so_15042791_15043019_0 . . . 332
13.231 so_19269438_19763712_0 . . . 332
13.232 so_43608823_43608969_2 . . . 332
13.233 so_29827028_29827190_1 . . . 332
13.234 so_5500057_30949326_0 . . . 332
13.235 so_44515378_44515398_2 . . . 332
13.236 so_31186912_31186912_0 . . . 333
13.237 so_31186912_31186912_2 . . . 333
13.238 so_31186912_31187310_0 . . . 333
13.239 so_31186912_31187310_1 . . . 333
13.240 so_31186912_31187855_0 . . . 333
13.241 so_25863485_25865297_2 . . . 333
13.242 so_27316607_27316852_0 . . . 333
13.243 so_21206359_21206750_1 . . . 333
13.244 so_27007831_27007831_0 . . . 333
13.245 so_11172389_11172417_0 . . . 333
13.246 so_17642624_17642624_0 . . . 333
13.247 so_39016352_39019826_0 . . . 333
13.248 so_40770829_40770829_2 . . . 333
13.249 so_21143835_21148117_0 . . . 333
13.250 so_10866895_10867345_0 . . . 333
13.251 so_5557392_5557436_0 . . . 333
13.252 so_33067600_33067600_34 . . . 333
13.253 so_33067600_33067600_13 . . . 333
13.254 so_33067600_33067600_26 . . . 333
13.255 so_22258831_22258831_0 . . . 333
13.256 so_22258831_22352944_0 . . . 333
13.257 so_11018810_11018810_1 . . . 333
13.258 so_11018810_11018810_2 . . . 333
13.259 so_11018810_11018810_0 . . . 333
13.260 so_26173403_26173614_0 . . . 333
13.261 so_8873048_8873237_0 . . . 333
13.262 so_8873048_8873237_1 . . . 333
13.263 so_35592392_35592630_1 . . . 333
13.264 so_14176320_14176320_0 . . . 333
13.265 so_33202400_33203861_4 . . . 334
13.266 so_27826824_27826995_1 . . . 334
13.267 so_19307578_19307651_0 . . . 334
13.268 so_26947599_26947994_0 . . . 334
13.269 so_33573164_33573346_0 . . . 334

13.270 so_18201645_18201697_0 . . . 334
13.271 so_24088148_24088197_3 . . . 334
13.272 so_24088148_24088197_1 . . . 334
13.273 so_25648460_25648645_0 . . . 334
13.274 so_33202400_33203861_0 . . . 334
13.275 so_27826824_27826995_2 . . . 334
13.276 so_25100855_25101393_0 . . . 334
13.277 so_26517376_26517912_0 . . . 334
13.278 so_41662100_41662321_0 . . . 334
13.279 so_18460289_18461082_1 . . . 334
13.280 so_42421458_42422428_4 . . . 334
13.281 so_46505988_46506254_1 . . . 334
13.282 so_18201645_18201697_1 . . . 334
13.283 so_24088148_24088197_2 . . . 334
13.284 so_24088148_24088197_4 . . . 334
13.285 so_40460715_40461307_0 . . . 334
13.286 so_41637152_41637450_1 . . . 334
13.287 so_36150408_36152300_0 . . . 334
13.288 so_36150408_36152300_1 . . . 334
13.289 so_21005172_21005338_0 . . . 334
13.290 so_26289918_26290347_0 . . . 334
13.291 so_24020132_24020803_1 . . . 334
13.292 so_13783932_13783939_0 . . . 334
13.293 so_47443172_47443373_0 . . . 334
13.294 so_44679200_44679576_1 . . . 334
13.295 so_31525445_31526379_0 . . . 334
13.296 so_14115516_14115575_0 . . . 334
13.297 so_14115516_14115575_2 . . . 334
13.298 so_14115516_14115575_3 . . . 334
13.299 so_14115516_14115575_1 . . . 334
13.300 so_44795721_44797625_1 . . . 334
13.301 so_25100855_25101393_2 . . . 334
13.302 so_14435619_14436738_0 . . . 334
13.303 so_27361230_27361572_0 . . . 334
13.304 so_26517376_26517912_1 . . . 334
13.305 so_26517376_26517912_2 . . . 334
13.306 so_39733786_39733908_0 . . . 334
13.307 so_18460289_18461082_9 . . . 334
13.308 so_18460289_18461082_6 . . . 334
13.309 so_18460289_18461082_8 . . . 334
13.310 so_18460289_18461082_10 . . . 334
13.311 so_18460289_18461082_7 . . . 334
13.312 so_18460289_18461082_5 . . . 334
13.313 so_18460289_18461082_11 . . . 334
13.314 so_18460289_18461082_3 . . . 334
13.315 so_18460289_18461082_4 . . . 334
13.316 so_18460289_18461082_2 . . . 334
13.317 so_41194123_41204286_0 . . . 334
13.318 so_9053299_9053364_0 . . . 334
13.319 so_25925551_26005482_5 . . . 334
13.320 so_27281500_27282211_0 . . . 334

13.321 so_27281500_27282211_1 . . . 334
13.322 so_40460715_40461307_1 . . . 334
13.323 so_23108590_23108966_1 . . . 334
13.324 so_23108590_23108966_0 . . . 334
13.325 so_13869150_13872036_1 . . . 334
13.326 so_30123338_30124715_0 . . . 334
13.327 so_2220230_2220254_1 . . . 335
13.328 so_39936865_39937432_1 . . . 335
13.329 so_43309255_43309255_1 . . . 335
13.330 so_43309255_43309255_2 . . . 335
13.331 so_43309255_43309255_0 . . . 335
13.332 so_21005172_21005338_2 . . . 335
13.333 so_14115516_14115575_4 . . . 335
13.334 so_44795721_44797625_0 . . . 335
13.335 so_36181895_36182152_0 . . . 335
13.336 so_26517376_26517912_3 . . . 335
13.337 so_46505988_46506254_2 . . . 335
13.338 so_41637152_41637450_3 . . . 335
13.339 so_3519282_3525172_0 . . . 335
13.340 so_25113661_25113863_3 . . . 336
13.341 so_41637152_41637450_4 . . . 336
13.342 so_45602368_45602368_2 . . . 336
13.343 so_45602368_45602368_0 . . . 336
13.344 so_2090867_2090885_2 . . . 336
13.345 so_37481767_37481993_3 . . . 336
13.346 so_37736780_37737049_5 . . . 336
13.347 so_37736780_37737049_4 . . . 336
13.348 so_37736780_37736780_0 . . . 336
13.349 so_30123338_30124715_2 . . . 336
13.350 so_39514765_39515239_0 . . . 336
13.351 so_41194123_41204286_1 . . . 336
13.352 so_27183046_27183415_0 . . . 336
13.353 so_30123338_30124715_1 . . . 336
13.354 so_13869150_13869150_0 . . . 336
13.355 so_23938354_23938813_0 . . . 337
13.356 so_13950556_13951621_0 . . . 337
13.357 so_11848031_11848814_1 . . . 337
13.358 so_35425488_35427581_2 . . . 337
13.359 so_3249147_3249392_4 . . . 337
13.360 so_40006925_40007311_0 . . . 337
13.361 so_29381171_29381331_2 . . . 337
13.362 so_46409188_46409188_2 . . . 337
13.363 so_33522985_33523669_0 . . . 337
13.364 so_40932844_40933118_1 . . . 337
13.365 so_26612554_26612762_2 . . . 337
13.366 so_41594186_41707564_9 . . . 337
13.367 so_29834308_29834426_2 . . . 337
13.368 so_39186010_39186556_0 . . . 337
13.369 so_12372442_12372442_0 . . . 337
13.370 so_12372442_12379512_0 . . . 337
13.371 so_15589842_15589842_0 . . . 337

13.372 so_27675893_27676077_4 . . . 337
13.373 so_34119876_34119876_0 . . . 337
13.374 so_1384007_1384044_2 . . . 337
13.375 so_15959179_15959298_1 . . . 337
13.376 so_35425488_35427581_3 . . . 337
13.377 so_6087729_6088828_0 . . . 337
13.378 so_12847272_12848540_4 . . . 337
13.379 so_12847272_12848540_5 . . . 337
13.380 so_12847272_12848540_3 . . . 337
13.381 so_1243962_1244346_1 . . . 337
13.382 so_17428394_17437848_0 . . . 337
13.383 so_8092670_8092754_0 . . . 337
13.384 so_46608738_46608866_1 . . . 337
13.385 so_42509344_42516715_1 . . . 337
13.386 so_41594186_41707564_2 . . . 337
13.387 so_25241516_25242107_5 . . . 337
13.388 so_30479923_30480820_1 . . . 337
13.389 so_30479923_30480820_0 . . . 337
13.390 so_30479923_30480820_2 . . . 337
13.391 so_18089988_18089988_7 . . . 337
13.392 so_1384007_1384044_3 . . . 337
13.393 so_44306607_44310738_3 . . . 337
13.394 so_25745759_25758367_3 . . . 337
13.395 so_39465404_39467467_2 . . . 337
13.396 so_25241516_25242107_13 . . . 337
13.397 so_18089988_18089988_5 . . . 337
13.398 so_18089988_18090865_1 . . . 337
13.399 so_18089988_18089988_6 . . . 337
13.400 so_1445274_1445327_4 . . . 337
13.401 so_38535292_38535667_5 . . . 337
13.402 so_38535292_38535667_4 . . . 337
13.403 so_3518145_3518343_0 . . . 337
13.404 so_40018076_40019058_2 . . . 337
13.405 so_42198789_42201902_0 . . . 337
13.406 so_39659011_39659230_1 . . . 337
13.407 so_28307887_28308012_0 . . . 337
13.408 so_25745759_25758367_2 . . . 337
13.409 so_39465404_39467467_1 . . . 337
13.410 so_26304949_26306505_0 . . . 337
13.411 so_25241516_25242022_6 . . . 337
13.412 so_25241516_25242107_8 . . . 337
13.413 so_25241516_25242107_6 . . . 337
13.414 so_15589842_15589842_3 . . . 338
13.415 so_27675893_27676077_2 . . . 338
13.416 so_3213591_3213820_1 . . . 338
13.417 so_3249147_3249527_5 . . . 338
13.418 so_39797134_42889321_1 . . . 338
13.419 so_18258745_18258986_0 . . . 338
13.420 so_40006925_40007311_2 . . . 338
13.421 so_31428980_31430017_1 . . . 338
13.422 so_31428980_31430017_2 . . . 338

13.423 so_29116751_29116751_0 . . . 338
13.424 so_29116751_29116751_1 . . . 338
13.425 so_39277869_39278520_1 . . . 338
13.426 so_38935256_38966560_2 . . . 338
13.427 so_31896298_31897188_3 . . . 338
13.428 so_31896298_31897188_5 . . . 338
13.429 so_31896298_31897188_2 . . . 338
13.430 so_31896298_31897188_4 . . . 338
13.431 so_13461027_13461382_0 . . . 338
13.432 so_28184984_28253270_2 . . . 338
13.433 so_37482588_37502709_1 . . . 338
13.434 so_25088827_25088924_1 . . . 338
13.435 so_25088827_25088924_0 . . . 338
13.436 so_26964221_26964314_1 . . . 338
13.437 so_26964221_26964314_2 . . . 338
13.438 so_26964221_26964314_3 . . . 338
13.439 so_26964221_26964314_4 . . . 338
13.440 so_34912221_34912221_1 . . . 338
13.441 so_34912221_34912592_0 . . . 338
13.442 so_34912221_34912221_0 . . . 338
13.443 so_34912221_34912221_2 . . . 338
13.444 so_26837275_26837580_1 . . . 338
13.445 so_26837275_26837580_2 . . . 338
13.446 so_26837275_26837580_0 . . . 338
13.447 so_27951537_27951602_0 . . . 338
13.448 so_27951537_27951602_1 . . . 338
13.449 so_27951537_27951602_2 . . . 338
13.450 so_30362362_30362664_0 . . . 338
13.451 so_30362362_30362664_1 . . . 338
13.452 so_20714273_20714357_4 . . . 338
13.453 so_20714273_20714357_2 . . . 338
13.454 so_20714273_20714357_5 . . . 338
13.455 so_20714273_20714357_3 . . . 338
13.456 so_20714273_20714357_6 . . . 338
13.457 so_20714273_20714357_1 . . . 338
13.458 so_31211021_31216487_5 . . . 338
13.459 so_31211021_31216487_3 . . . 338
13.460 so_31211021_31216487_2 . . . 338
13.461 so_31211021_31216487_1 . . . 338
13.462 so_29763339_29763864_0 . . . 338
13.463 so_37024314_37025347_1 . . . 338
13.464 so_37024314_37025347_0 . . . 338
13.465 so_43672392_43720636_2 . . . 338
13.466 so_13693871_13694435_1 . . . 338
13.467 so_27144054_27144307_1 . . . 338
13.468 so_27144054_27144307_0 . . . 338
13.469 so_38131945_38131963_8 . . . 338
13.470 so_38131945_38131963_7 . . . 338
13.471 so_9723164_9723448_2 . . . 338
13.472 so_9723164_9723448_0 . . . 338
13.473 so_9723164_9723448_1 . . . 338

13.474 so_42586836_42587200_4 . . . 338
13.475 so_42586836_42587200_5 . . . 338
13.476 so_26137325_26137910_1 . . . 338
13.477 so_2357452_2357688_8 . . . 338
13.478 so_33970892_33978450_1 . . . 338
13.479 so_33970892_33978450_0 . . . 338
13.480 so_26414443_26414748_1 . . . 338
13.481 so_6744524_6746173_0 . . . 338
13.482 so_18215277_18216193_0 . . . 338
13.483 so_17979857_17980055_2 . . . 338
13.484 so_17979857_17980055_1 . . . 338
13.485 so_17979857_17980055_0 . . . 338
13.486 so_35273025_35273245_4 . . . 338
13.487 so_35273025_35273245_3 . . . 338
13.488 so_15641318_15641365_1 . . . 338
13.489 so_15641318_15641365_2 . . . 338
13.490 so_28058371_28058430_3 . . . 338
13.491 so_28058371_28058430_2 . . . 338
13.492 so_31594713_31594816_2 . . . 338
13.493 so_31594713_31594816_1 . . . 338
13.494 so_15852528_15852626_2 . . . 338
13.495 so_15852528_15852626_1 . . . 338
13.496 so_15852528_15852626_3 . . . 338
13.497 so_7132879_7134630_0 . . . 338
13.498 so_21825933_21836197_0 . . . 338
13.499 so_12444503_12630072_0 . . . 338
13.500 so_30108126_30108217_1 . . . 338
13.501 so_30108126_30108217_0 . . . 338
13.502 so_45995902_45996037_5 . . . 338
13.503 so_31728686_31728949_1 . . . 338
13.504 so_9157041_9158170_0 . . . 338
13.505 so_9157041_9157041_0 . . . 338
13.506 so_40784322_40784990_3 . . . 338
13.507 so_40784322_40784990_1 . . . 338
13.508 so_40784322_40784990_2 . . . 338
13.509 so_27573928_27574224_0 . . . 338
13.510 so_26426222_26426367_1 . . . 338
13.511 so_33298375_33298417_0 . . . 338
13.512 so_33298375_33298417_2 . . . 338
13.513 so_33298375_33298417_1 . . . 338
13.514 so_38809275_38809409_0 . . . 338
13.515 so_24718555_24719009_0 . . . 338
13.516 so_44306607_44310738_5 . . . 338
13.517 so_32850179_32850659_2 . . . 338
13.518 so_32850179_32850659_1 . . . 338
13.519 so_32270244_32270347_2 . . . 338
13.520 so_32270244_32270723_1 . . . 338
13.521 so_32270244_32270347_3 . . . 338
13.522 so_38061779_38065158_1 . . . 338
13.523 so_32144976_32145795_0 . . . 338
13.524 so_35336990_35336990_1 . . . 338

13.525 so_34622076_34839362_0 . . . 338
13.526 so_26947704_26949099_3 . . . 338
13.527 so_26947704_26949099_0 . . . 338
13.528 so_26947704_26949099_2 . . . 338
13.529 so_26947704_26949099_1 . . . 338
13.530 so_26947704_26949366_2 . . . 338
13.531 so_33202400_33203861_1 . . . 338
13.532 so_18089988_18090865_8 . . . 338
13.533 so_46409188_46409188_1 . . . 338
13.534 so_43802054_43802594_0 . . . 338
13.535 so_27078793_27080014_0 . . . 338
13.536 so_43793941_43794166_2 . . . 338
13.537 so_12436557_12436695_0 . . . 339
13.538 so_33202400_33203861_2 . . . 339
13.539 so_18089988_18090865_9 . . . 339
13.540 so_31186912_31187855_1 . . . 339
13.541 so_19083709_19083981_0 . . . 339
13.542 so_21974678_21974770_3 . . . 339
13.543 so_21974678_21974770_1 . . . 339
13.544 so_21974678_21974770_2 . . . 339
13.545 so_21974678_21974770_0 . . . 339
13.546 so_4706677_4706677_0 . . . 339
13.547 so_12414804_12414835_0 . . . 339
13.548 so_29516653_29516807_0 . . . 339
13.549 so_43802054_43802594_2 . . . 339
13.550 so_43802054_43802594_1 . . . 339
13.551 so_43802054_43802594_4 . . . 339
13.552 so_43802054_43802594_3 . . . 339
13.553 so_12109471_12110635_0 . . . 339
13.554 so_10866895_10867345_2 . . . 339
13.555 so_12372442_12373830_0 . . . 339
13.556 so_33202400_33203861_3 . . . 339
13.557 so_18089988_18090865_10 . . . 339
13.558 so_31186912_31187855_2 . . . 339
13.559 so_31730222_31730222_0 . . . 339
13.560 so_31728686_31729647_2 . . . 339
13.561 so_13434672_13543853_3 . . . 339
13.562 so_28074071_28074071_0 . . . 339
13.563 so_18899383_18899438_1 . . . 339
13.564 so_15589842_15590092_2 . . . 339
13.565 so_27301135_27301428_1 . . . 339
13.566 so_31896298_31897188_12 . . . 339
13.567 so_31896298_31897188_13 . . . 339
13.568 so_31126559_31128008_0 . . . 339
13.569 so_8986558_8987000_1 . . . 339
13.570 so_24286523_24287590_3 . . . 339
13.571 so_24286523_24287590_2 . . . 339
13.572 so_24286523_24287590_1 . . . 339
13.573 so_24286523_24287590_5 . . . 339
13.574 so_15578298_15580072_4 . . . 339
13.575 so_15578298_15580072_3 . . . 339

13.576 so_15578298_15580072_5 . . . 339
13.577 so_15578298_15580072_7 . . . 339
13.578 so_18089988_18090865_11 . . . 339
13.579 so_18089988_18090865_12 . . . 339
13.580 so_18089988_18090865_13 . . . 339
13.581 so_7034297_7034414_1 . . . 339
13.582 so_14666219_14666246_0 . . . 339
13.583 so_39286114_39287634_3 . . . 339
13.584 so_29961462_29962030_0 . . . 339
13.585 so_27679002_27679053_4 . . . 339
13.586 so_31520540_31520780_0 . . . 339
13.587 so_31520540_31520796_0 . . . 339
13.588 so_24029213_24029380_0 . . . 339
13.589 so_37354243_37354609_0 . . . 339
13.590 so_36603628_36603734_2 . . . 339
13.591 so_43304188_44145375_3 . . . 339
13.592 so_32442699_32443367_1 . . . 339
13.593 so_32442699_32443367_0 . . . 339
13.594 so_598913_604022_0 . . . 339
13.595 so_43732122_43732722_2 . . . 339
13.596 so_28168290_28170899_0 . . . 339
13.597 so_31475016_31475554_1 . . . 339
13.598 so_31475016_31475554_2 . . . 339
13.599 so_31475016_31475554_5 . . . 339
13.600 so_31475016_31475554_3 . . . 339
13.601 so_31475016_31475554_0 . . . 339
13.602 so_34119876_34120764_0 . . . 339
13.603 so_34119876_34120764_1 . . . 339
13.604 so_42244250_42249785_0 . . . 339
13.605 so_21469135_21469596_1 . . . 339
13.606 so_28824330_28824795_0 . . . 339
13.607 so_25836109_25836942_0 . . . 339
13.608 so_25836109_25836942_1 . . . 339
13.609 so_20057403_20057957_0 . . . 339
13.610 so_11545316_11545508_4 . . . 339
13.611 so_11545316_11545508_1 . . . 339
13.612 so_23423966_23424604_9 . . . 339
13.613 so_23423966_23424604_7 . . . 339
13.614 so_23423966_23424604_3 . . . 339
13.615 so_23423966_23424604_4 . . . 339
13.616 so_23423966_23424604_2 . . . 339
13.617 so_23423966_23424604_14 . . . 339
13.618 so_23423966_23424604_5 . . . 339
13.619 so_23423966_23424604_12 . . . 339
13.620 so_23423966_23424604_11 . . . 339
13.621 so_23423966_23424604_10 . . . 339
13.622 so_23423966_23424604_1 . . . 339
13.623 so_46019548_46019670_0 . . . 339
13.624 so_15586526_15586567_2 . . . 339
13.625 so_15586526_15586567_1 . . . 339
13.626 so_15586526_15586567_3 . . . 339

13.627 so_15586526_15586567_5 . . . 339
13.628 so_15028835_15029228_3 . . . 339
13.629 so_15028835_15029228_5 . . . 339
13.630 so_14987326_14990231_4 . . . 339
13.631 so_14987326_14990231_0 . . . 339
13.632 so_14987326_14990231_1 . . . 339
13.633 so_14987326_14990231_2 . . . 339
13.634 so_8445563_8447183_1 . . . 339
13.635 so_8445563_8447183_2 . . . 339
13.636 so_8445563_8447183_3 . . . 339
13.637 so_44338584_44339464_1 . . . 339
13.638 so_8914986_8917084_5 . . . 339
13.639 so_8914986_8916460_0 . . . 339
13.640 so_31814214_31814278_0 . . . 339
13.641 so_11296389_12610403_7 . . . 339
13.642 so_11296389_12610403_3 . . . 339
13.643 so_11296389_12610403_4 . . . 339
13.644 so_11296389_12610403_5 . . . 339
13.645 so_12419384_12419609_3 . . . 339
13.646 so_12419384_12419609_4 . . . 339
13.647 so_11555950_11555950_1 . . . 339
13.648 so_28243371_28243509_0 . . . 339
13.649 so_17130607_17130847_4 . . . 339
13.650 so_1177739_1177988_0 . . . 339
13.651 so_1177739_1177988_1 . . . 339
13.652 so_25825867_25828064_2 . . . 339
13.653 so_25825867_25828064_1 . . . 339
13.654 so_42573771_42574950_4 . . . 339
13.655 so_42573771_42574950_3 . . . 339
13.656 so_15601057_15601166_0 . . . 339
13.657 so_15601057_15601166_1 . . . 339
13.658 so_15601057_15601166_2 . . . 339
13.659 so_31047860_31048865_6 . . . 339
13.660 so_31047860_31048865_0 . . . 339
13.661 so_31047860_31048865_5 . . . 339
13.662 so_31391172_31391456_0 . . . 339
13.663 so_31391172_31391456_4 . . . 339
13.664 so_31391172_31391456_2 . . . 339
13.665 so_31391172_31391456_1 . . . 339
13.666 so_28079317_28086248_1 . . . 339
13.667 so_9622089_9636595_1 . . . 339
13.668 so_45593044_45593724_0 . . . 339
13.669 so_30571771_30571897_0 . . . 339
13.670 so_33095439_33095469_3 . . . 339
13.671 so_4263748_4264747_0 . . . 339
13.672 so_17453738_28557688_3 . . . 339
13.673 so_12748212_12748299_0 . . . 339
13.674 so_29090692_29090895_0 . . . 339
13.675 so_32537994_32539424_0 . . . 339
13.676 so_16635822_16636025_1 . . . 339
13.677 so_16635822_16636025_2 . . . 339

13.678 so_16635822_16636025_0 . . . 339
13.679 so_47415476_47415915_2 . . . 339
13.680 so_47415476_47415915_1 . . . 339
13.681 so_47415476_47415915_3 . . . 339
13.682 so_25047109_25047700_1 . . . 339
13.683 so_29961100_29961422_4 . . . 339
13.684 so_29961100_29961422_0 . . . 339
13.685 so_29961100_29961422_5 . . . 339
13.686 so_3898757_3898787_1 . . . 339
13.687 so_9728693_9728764_2 . . . 339
13.688 so_31464666_31465009_0 . . . 339
13.689 so_38927019_38927195_4 . . . 339
13.690 so_38927019_38927195_1 . . . 339
13.691 so_46116173_46116173_0 . . . 339
13.692 so_46108455_46108512_0 . . . 339
13.693 so_28308599_28309212_2 . . . 339
13.694 so_28308599_28308748_1 . . . 339
13.695 so_32338988_32339430_1 . . . 339
13.696 so_17332320_17333188_0 . . . 339
13.697 so_17332320_17333188_1 . . . 339
13.698 so_27078793_27080014_5 . . . 339
13.699 so_25395056_25395056_2 . . . 339
13.700 so_25395056_26110725_5 . . . 339
13.701 so_25395056_26110725_3 . . . 339
13.702 so_25395056_26110725_2 . . . 339
13.703 so_30064094_30064173_1 . . . 339
13.704 so_26000553_26001996_0 . . . 339
13.705 so_28224259_28224463_2 . . . 339
13.706 so_28224259_28224463_1 . . . 339
13.707 so_28224259_28224463_0 . . . 339
13.708 so_2169932_2170050_3 . . . 339
13.709 so_26858076_26858409_5 . . . 339
13.710 so_26858076_26858409_2 . . . 339
13.711 so_32373979_32375678_1 . . . 339
13.712 so_26724674_26725089_0 . . . 339
13.713 so_26724674_26725089_1 . . . 339
13.714 so_6115204_6115281_1 . . . 339
13.715 so_15497004_15497032_4 . . . 339
13.716 so_15497004_15497032_0 . . . 339
13.717 so_15497004_15497032_1 . . . 339
13.718 so_15497004_15497032_2 . . . 339
13.719 so_45353121_45700164_0 . . . 339
13.720 so_45353121_45353121_2 . . . 339
13.721 so_39355629_39356164_0 . . . 339
13.722 so_31534195_31534330_1 . . . 339
13.723 so_46997165_46999091_7 . . . 339
13.724 so_40564690_40564690_0 . . . 339
13.725 so_15208198_15208571_4 . . . 339
13.726 so_15208198_15208300_2 . . . 339
13.727 so_15208198_15208571_5 . . . 339
13.728 so_15208198_15208571_2 . . . 339

13.729 so_15208198_15208606_1 . . . 339
13.730 so_15208198_15208300_0 . . . 339
13.731 so_15208198_15208571_1 . . . 339
13.732 so_15208198_15208571_3 . . . 339
13.733 so_15208198_15208571_7 . . . 339
13.734 so_25745759_25758367_11 . . . 339
13.735 so_25745759_25758367_10 . . . 339
13.736 so_25745759_25758367_9 . . . 339
13.737 so_23804312_23806297_2 . . . 339
13.738 so_23804312_23806297_1 . . . 339
13.739 so_23804312_23806297_5 . . . 339
13.740 so_23804312_23806297_3 . . . 339
13.741 so_23804312_23806297_4 . . . 339
13.742 so_23804312_23806297_7 . . . 339
13.743 so_39465404_39467467_4 . . . 339
13.744 so_38356069_38356176_0 . . . 339
13.745 so_35546335_35546705_0 . . . 339
13.746 so_43645582_43645824_0 . . . 339
13.747 so_26317798_26318330_1 . . . 339
13.748 so_26317798_26318330_0 . . . 339
13.749 so_21210360_21210565_3 . . . 339
13.750 so_3519282_3525172_1 . . . 339
13.751 so_3519282_3525172_2 . . . 339
13.752 so_38033124_38033748_1 . . . 339
13.753 so_10278144_10278594_1 . . . 339
13.754 so_10278144_10278594_0 . . . 339
13.755 so_10278144_10278594_3 . . . 339
13.756 so_25241516_25242022_8 . . . 339
13.757 so_16764285_16764306_0 . . . 339
13.758 so_16764285_16764306_4 . . . 339
13.759 so_16764285_16764306_2 . . . 339
13.760 so_16764285_16764306_1 . . . 339
13.761 so_29324762_29324919_1 . . . 339
13.762 so_29324762_29324919_2 . . . 339
13.763 so_34899524_34899663_0 . . . 339
13.764 so_3431735_3431811_1 . . . 339
13.765 so_3431735_3431811_3 . . . 339
13.766 so_3431735_3431811_0 . . . 339
13.767 so_30077526_30078292_1 . . . 339
13.768 so_30077526_30078294_0 . . . 339
13.769 so_30077526_30078292_0 . . . 339
13.770 so_16922121_16922652_1 . . . 340
13.771 so_11545316_11545508_2 . . . 340
13.772 so_23423966_23424604_8 . . . 340
13.773 so_14987326_14990231_5 . . . 340
13.774 so_8914986_8917084_6 . . . 340
13.775 so_3898757_3898787_2 . . . 340
13.776 so_27078793_27080014_4 . . . 340
13.777 so_5552455_5552911_0 . . . 340
13.778 so_10866895_10867345_3 . . . 340
13.779 so_29324762_29324919_4 . . . 340

13.780 so_12372442_12373830_1 . . . 341
13.781 so_16602175_16602198_0 . . . 341
13.782 so_27054485_27054786_0 . . . 341
13.783 so_27054485_27054485_0 . . . 341
13.784 so_27078793_27080014_1 . . . 341
13.785 so_27032037_27032037_0 . . . 341
13.786 so_39465404_39467467_3 . . . 341
13.787 so_27074630_27074630_0 . . . 341
13.788 so_18489778_18490078_0 . . . 341
13.789 so_25552822_25553119_5 . . . 341
13.790 so_27032037_27032037_1 . . . 341
13.791 so_41594186_41707564_4 . . . 341
13.792 so_28338760_28338958_3 . . . 341
13.793 so_42586836_42587200_8 . . . 341
13.794 so_36910778_36910778_2 . . . 341
13.795 so_36910778_36910778_3 . . . 341
13.796 so_36910778_36910778_4 . . . 341
13.797 so_27118985_27118985_0 . . . 341
13.798 so_32469979_32475135_3 . . . 341
13.799 so_12444503_12630072_1 . . . 341
13.800 so_27233505_27234023_1 . . . 341
13.801 so_27233505_27233505_0 . . . 341
13.802 so_27233505_27234023_2 . . . 341
13.803 so_19745768_19759146_0 . . . 341
13.804 so_12677711_12732730_3 . . . 341
13.805 so_12677711_12677711_0 . . . 341
13.806 so_12677711_12732730_4 . . . 341
13.807 so_12677711_12732730_6 . . . 341
13.808 so_41594186_41707564_5 . . . 341
13.809 so_41594186_41707564_6 . . . 341
13.810 so_32491876_32491876_3 . . . 341
13.811 so_32491876_32491876_6 . . . 341
13.812 so_32491876_32491876_2 . . . 341
13.813 so_32491876_32491876_4 . . . 341
13.814 so_30479923_30480820_5 . . . 341
13.815 so_46409188_46409188_3 . . . 342
13.816 so_4704567_4704939_0 . . . 342
13.817 so_15586526_15586575_4 . . . 342
13.818 so_2748866_2748896_0 . . . 342
13.819 so_34490247_34490537_0 . . . 342
13.820 so_37166131_37166556_3 . . . 342
13.821 so_27328876_27328876_0 . . . 342
13.822 so_38927019_38927019_3 . . . 342
13.823 so_26754107_26754423_3 . . . 342
13.824 so_27078793_27080014_2 . . . 342
13.825 so_28405921_28406267_0 . . . 342
13.826 so_32491876_32502276_1 . . . 342
13.827 so_32491876_32502276_0 . . . 342
13.828 so_34490247_34490537_1 . . . 342
13.829 so_8771409_8771433_3 . . . 342
13.830 so_28405921_28406267_1 . . . 342

13.831 so_28338760_28338958_6 . . . 342
13.832 so_8771409_8771433_4 . . . 342
13.833 so_25385464_25386824_0 . . . 342
13.834 so_8771409_8771433_5 . . . 342
13.835 so_25395056_26110725_7 . . . 342
13.836 so_25395056_25395056_0 . . . 342
13.837 so_41594186_41688134_1 . . . 342
13.838 so_25552822_25553119_6 . . . 343
13.839 so_25552822_25553119_7 . . . 343
13.840 so_898955_898975_1 . . . 343
13.841 so_3518145_3518343_1 . . . 343
13.842 so_44815376_44815571_1 . . . 343
13.843 so_38927019_38927195_0 . . . 343
13.844 so_27078793_27080014_3 . . . 343
13.845 so_14915216_14915364_0 . . . 343
13.846 so_15586526_15586575_5 . . . 343
13.847 so_32274802_32274900_0 . . . 343
13.848 so_35425488_35427581_0 . . . 343
13.849 so_6401818_6402127_0 . . . 343
13.850 so_9619659_9620134_1 . . . 343
13.851 so_27340009_27340009_0 . . . 343
13.852 so_25113661_25113863_0 . . . 343
13.853 so_8610511_8610842_0 . . . 343
13.854 so_23610117_23610117_0 . . . 343
13.855 so_28338760_28338958_7 . . . 343
13.856 so_26954276_26956094_0 . . . 343
13.857 so_26954276_26955983_6 . . . 343
13.858 so_25241516_25242107_14 . . . 343
13.859 so_25241516_25242107_15 . . . 343
13.860 so_22444415_22490313_0 . . . 344
13.861 so_25113661_25113863_1 . . . 344
13.862 so_25395056_26110725_6 . . . 344
13.863 so_26954276_26955983_7 . . . 344
13.864 so_3888082_3888082_0 . . . 344
13.865 so_15852528_15852626_0 . . . 344
13.866 so_26754107_26754423_1 . . . 344
13.867 so_26317798_26318330_4 . . . 344
13.868 so_2749263_2749263_0 . . . 344
13.869 so_19083709_19083981_4 . . . 344
13.870 so_23804312_23806297_11 . . . 344
13.871 so_27301135_27301428_2 . . . 344
13.872 so_32368890_32371173_0 . . . 344
13.873 so_36696312_36696331_0 . . . 344
13.874 so_31186912_31187855_3 . . . 344
13.875 so_41595749_41596032_0 . . . 344
13.876 so_41505121_41505189_1 . . . 344
13.877 so_41505121_41505189_2 . . . 344
13.878 so_36450983_36451214_1 . . . 344
13.879 so_26113016_26113270_1 . . . 344
13.880 so_33074102_33074102_2 . . . 344
13.881 so_33074102_33074102_3 . . . 344

13.882 so_13798897_13799098_0 . . . 344
13.883 so_46409188_46409188_4 . . . 344
13.884 so_26754107_26754423_0 . . . 344
13.885 so_26754107_26754423_2 . . . 344
13.886 so_18409239_18411541_1 . . . 344
13.887 so_31614816_31636049_0 . . . 344
13.888 so_26724674_26725089_2 . . . 344
13.889 so_28405921_28406617_8 . . . 344
13.890 so_27603829_27603921_0 . . . 344
13.891 so_42613660_42614857_0 . . . 345
13.892 so_43672392_43720636_3 . . . 345
13.893 so_2748866_2749602_0 . . . 345
13.894 so_20973417_21044373_2 . . . 345
13.895 so_33074102_33074102_4 . . . 345
13.896 so_2169932_2170050_1 . . . 345
13.897 so_15589842_15590092_1 . . . 345
13.898 so_30559817_30561123_0 . . . 345
13.899 so_2748866_2749602_1 . . . 345
13.900 so_34160614_34163861_0 . . . 345
13.901 so_27074630_27074757_0 . . . 345
13.902 so_16926855_16927215_0 . . . 345
13.903 so_19256322_19257938_0 . . . 345
13.904 so_43672392_43720636_0 . . . 345
13.905 so_2357452_2357688_2 . . . 345
13.906 so_33074102_33074102_0 . . . 345
13.907 so_35336990_35337463_0 . . . 345
13.908 so_34912221_34912221_4 . . . 345
13.909 so_31211021_31216487_4 . . . 345
13.910 so_2357452_2357688_9 . . . 345
13.911 so_27397701_27399225_1 . . . 345
13.912 so_38795374_38796733_0 . . . 345
13.913 so_32586871_32587320_0 . . . 345
13.914 so_31728686_31729647_1 . . . 345
13.915 so_28074071_28074354_0 . . . 345
13.916 so_16939471_16939798_2 . . . 346
13.917 so_35114085_35114085_0 . . . 346
13.918 so_31896298_31897188_6 . . . 346
13.919 so_43672392_43720636_1 . . . 346
13.920 so_43672392_43720636_4 . . . 346
13.921 so_42586836_42587200_6 . . . 346
13.922 so_2357452_2357688_3 . . . 346
13.923 so_44573671_44573909_0 . . . 346
13.924 so_35336990_35337463_1 . . . 346
13.925 so_16985687_16985687_3 . . . 347
13.926 so_20973417_21044373_3 . . . 347
13.927 so_26788402_26788402_1 . . . 347
13.928 so_31278377_31278474_2 . . . 347
13.929 so_45834990_45835907_0 . . . 347
13.930 so_26699720_26700151_3 . . . 347
13.931 so_23882409_23883035_11 . . . 347
13.932 so_26256174_26256294_6 . . . 347

13.933 so_15126495_15126673_2 . . . 347
13.934 so_17257828_17257838_4 . . . 347
13.935 so_20973417_21044373_1 . . . 347
13.936 so_21120788_21177636_0 . . . 347
13.937 so_33074102_33074102_1 . . . 347
13.938 so_44573671_44573909_2 . . . 347
13.939 so_31896298_31897188_8 . . . 347
13.940 so_31896298_31897188_9 . . . 347
13.941 so_42586836_42587200_7 . . . 347
13.942 so_33479555_33479868_2 . . . 347
13.943 so_33479555_33479868_0 . . . 347
13.944 so_44573671_44573909_4 . . . 347
13.945 so_44573671_44573909_6 . . . 347
13.946 so_27951537_27951602_3 . . . 348
13.947 so_25241516_25242107_16 . . . 348
13.948 so_31896298_31897188_14 . . . 348
13.949 so_31896298_31897188_15 . . . 348
13.950 so_24503031_24503406_3 . . . 348
13.951 so_24503031_24503406_4 . . . 348
13.952 so_24503031_24503406_2 . . . 348
13.953 so_32442699_32443367_4 . . . 348
13.954 so_32442699_32443367_3 . . . 348
13.955 so_32442699_32443367_2 . . . 348
13.956 so_19083709_19083981_5 . . . 348
13.957 so_15622760_15622820_0 . . . 348
13.958 so_1092714_47414201_2 . . . 348
13.959 so_1092714_47414201_1 . . . 348
13.960 so_1092714_47414201_3 . . . 348
13.961 so_8914986_8917084_4 . . . 348
13.962 so_8359201_8359260_1 . . . 348
13.963 so_29516653_29516941_0 . . . 348
13.964 so_29516653_29516897_0 . . . 348
13.965 so_25764681_25765295_0 . . . 348
13.966 so_36268560_36268747_3 . . . 348
13.967 so_36268560_36268747_1 . . . 348
13.968 so_36268560_36268747_2 . . . 348
13.969 so_15032044_15032114_3 . . . 348
13.970 so_15032044_15032114_1 . . . 348
13.971 so_15032044_15032114_2 . . . 348
13.972 so_4316316_4322041_7 . . . 348
13.973 so_4316316_4322041_6 . . . 348
13.974 so_4316316_4322041_8 . . . 348
13.975 so_16319813_16319857_0 . . . 348
13.976 so_30996948_30997325_0 . . . 348
13.977 so_27341171_27341212_0 . . . 348
13.978 so_14770252_14770297_0 . . . 348
13.979 so_23804312_23806297_12 . . . 348
13.980 so_1149109_1149311_0 . . . 348
13.981 so_13554136_13554155_0 . . . 348
13.982 so_28338760_28338958_2 . . . 348
13.983 so_28338760_28338958_1 . . . 348

13.984 so_28338760_28338958_0 . . . 348
13.985 so_24876550_24876671_3 . . . 348
13.986 so_24876550_24876671_2 . . . 348
13.987 so_15589842_15590092_4 . . . 348
13.988 so_31126559_31128008_1 . . . 348
13.989 so_36592933_36607925_0 . . . 348
13.990 so_8986558_8987000_0 . . . 348
13.991 so_8986558_8987000_2 . . . 348
13.992 so_31186912_31187855_5 . . . 348
13.993 so_27679002_27679053_3 . . . 348
13.994 so_24029213_24029380_2 . . . 348
13.995 so_33631895_33632903_0 . . . 348
13.996 so_37354243_37354609_1 . . . 348
13.997 so_41505121_41505189_4 . . . 348
13.998 so_41505121_41505189_5 . . . 348
13.999 so_46579016_46579091_0 . . . 348
13.1000 so_21469135_21469596_2 . . . 348
13.1001 so_11545316_11545508_6 . . . 348
13.1002 so_1384007_1384044_4 . . . 348
13.1003 so_33834573_33848588_4 . . . 348
13.1004 so_18625172_18625172_2 . . . 348
13.1005 so_18625172_18625389_0 . . . 348
13.1006 so_10743106_10743255_1 . . . 348
13.1007 so_10743106_10743255_0 . . . 348
13.1008 so_10743106_10743255_2 . . . 348
13.1009 so_34490247_34490489_0 . . . 348
13.1010 so_11555950_11555950_2 . . . 348
13.1011 so_28243371_28243509_1 . . . 348
13.1012 so_28243371_28243509_2 . . . 348
13.1013 so_17130607_17130847_5 . . . 348
13.1014 so_3518145_3518343_2 . . . 348
13.1015 so_47608388_47610648_2 . . . 348
13.1016 so_42573771_42574950_2 . . . 348
13.1017 so_42573771_42574950_1 . . . 348
13.1018 so_31047860_31048865_1 . . . 348
13.1019 so_31047860_31048865_3 . . . 348
13.1020 so_31047860_31048865_4 . . . 348
13.1021 so_31047860_31048865_2 . . . 348
13.1022 so_26168013_26171543_0 . . . 348
13.1023 so_27397701_27399225_3 . . . 348
13.1024 so_46015090_46015090_0 . . . 348
13.1025 so_25624303_25624367_1 . . . 348
13.1026 so_4263748_4263748_2 . . . 348
13.1027 so_4263748_4265666_0 . . . 348
13.1028 so_36070303_36086450_0 . . . 348
13.1029 so_29961100_29961422_2 . . . 348
13.1030 so_29961100_29961422_1 . . . 348
13.1031 so_32586871_32587320_1 . . . 348
13.1032 so_31464666_31465009_1 . . . 348
13.1033 so_38927019_38927195_6 . . . 348
13.1034 so_38927019_38927195_3 . . . 348

13.1035 so_18409239_18411541_0 . . . 348
13.1036 so_46116173_46116173_2 . . . 348
13.1037 so_46116173_46122538_0 . . . 348
13.1038 so_37103718_37103824_1 . . . 348
13.1039 so_46108455_46108512_4 . . . 348
13.1040 so_46108455_46108512_2 . . . 348
13.1041 so_11752307_11752401_1 . . . 348
13.1042 so_35114085_35114085_1 . . . 348
13.1043 so_32338988_32339430_3 . . . 348
13.1044 so_32338988_32339430_2 . . . 348
13.1045 so_26573100_26573271_0 . . . 348
13.1046 so_30064094_30064173_0 . . . 348
13.1047 so_5347444_5352236_0 . . . 348
13.1048 so_2169932_2170050_2 . . . 348
13.1049 so_2169932_2170050_4 . . . 348
13.1050 so_31534195_31534330_3 . . . 348
13.1051 so_31534195_31534330_0 . . . 348
13.1052 so_46997165_46999091_9 . . . 348
13.1053 so_40564690_40564690_2 . . . 348
13.1054 so_11880585_11880679_0 . . . 348
13.1055 so_33088586_33088753_1 . . . 348
13.1056 so_37218249_37219591_0 . . . 348
13.1057 so_26524250_26524503_0 . . . 348
13.1058 so_3519282_3519584_0 . . . 348
13.1059 so_3519282_3519584_1 . . . 348
13.1060 so_25241516_25242107_17 . . . 348
13.1061 so_17819109_17819109_1 . . . 348
13.1062 so_29324762_29324919_5 . . . 348
13.1063 so_23377647_23393808_2 . . . 348
13.1064 so_23377647_23393808_1 . . . 348
13.1065 so_17961719_17961890_1 . . . 348
13.1066 so_25552822_25553119_9 . . . 350
13.1067 so_25552822_25553119_8 . . . 350
13.1068 so_18089988_18090865_14 . . . 350
13.1069 so_11650868_11651082_4 . . . 350
13.1070 so_11650868_11651082_2 . . . 350
13.1071 so_32522701_32523222_1 . . . 350
13.1072 so_34336024_34336157_1 . . . 350
13.1073 so_4640543_4640680_0 . . . 350
13.1074 so_35536146_35538979_2 . . . 350
13.1075 so_37166131_37166556_1 . . . 350
13.1076 so_38927019_38927019_1 . . . 350
13.1077 so_14745825_14746323_0 . . . 350
13.1078 so_27226680_27226680_2 . . . 350
13.1079 so_27226680_27226680_3 . . . 350
13.1080 so_27226680_27226680_5 . . . 350
13.1081 so_14206403_14208457_1 . . . 350
13.1082 so_6722333_6722355_0 . . . 350
13.1083 so_17819109_17819235_1 . . . 350
13.1084 Address of overloaded function . . . 351
13.1085 so_24434157_24630425_3 . . . 351

13.1086 so_19269438_19819649_0 . . . 351
13.1087 so_19269438_19737662_0 . . . 351
13.1088 so_14945564_14945668_1 . . . 351
13.1089 so_2065228_2065497_1 . . . 351
13.1090 so_2065228_2065497_0 . . . 351
13.1091 so_18355814_18361773_0 . . . 351
13.1092 so_6007563_6008660_1 . . . 351
13.1093 so_6007563_6008660_0 . . . 351
13.1094 so_15959179_15959298_0 . . . 351
13.1095 so_15959179_15959298_5 . . . 351
13.1096 so_30130803_30131242_3 . . . 351
13.1097 so_30130803_30131242_1 . . . 351
13.1098 so_30130803_30131242_2 . . . 351
13.1099 so_40343663_40350274_3 . . . 351
13.1100 so_26792376_26792689_1 . . . 351
13.1101 so_26792376_26792689_0 . . . 351
13.1102 so_27970737_27970833_1 . . . 351
13.1103 so_30227277_30227743_0 . . . 351
13.1104 so_16542026_16542381_1 . . . 351
13.1105 so_16542026_16543835_2 . . . 351
13.1106 so_35543211_35543548_0 . . . 351
13.1107 so_1301380_1301465_1 . . . 351
13.1108 so_28008073_28008120_0 . . . 351
13.1109 so_42509344_42516715_8 . . . 351
13.1110 so_42509344_42516715_5 . . . 351
13.1111 so_42509344_42516715_0 . . . 351
13.1112 so_12358843_40826136_3 . . . 351
13.1113 so_1424261_1424314_0 . . . 351
13.1114 so_1424261_1424314_1 . . . 351
13.1115 so_20652424_20652665_2 . . . 351
13.1116 so_20652424_20652665_0 . . . 351
13.1117 so_24434157_24630425_4 . . . 351
13.1118 so_40343663_40350274_4 . . . 351
13.1119 so_6798742_6800024_0 . . . 351
13.1120 so_41175957_41176278_1 . . . 351
13.1121 so_40343663_40350274_5 . . . 352
13.1122 Overloaded operators . . . 352
13.1123 so_46624005_47250175_8 . . . 352
13.1124 so_36181895_36182442_1 . . . 352
13.1125 so_28366444_28368013_4 . . . 352
13.1126 so_28366444_28368013_0 . . . 352
13.1127 so_28794443_28794594_1 . . . 352
13.1128 so_28794443_28794594_3 . . . 353
13.1129 so_33097853_33097909_0 . . . 353
13.1130 so_32103998_32104249_0 . . . 353
13.1131 so_13360640_13360673_1 . . . 353
13.1132 so_24168137_24168426_0 . . . 353
13.1133 so_7225962_7226010_1 . . . 353
13.1134 so_5063924_5063957_0 . . . 353
13.1135 so_1108360_1108511_0 . . . 353
13.1136 so_1108360_1108360_1 . . . 353

13.1137 so_12919474_12919474_3 . . . 353
13.1138 so_32776445_32776545_0 . . . 353
13.1139 so_30898602_30900449_3 . . . 353
13.1140 so_30898602_30900449_5 . . . 353
13.1141 so_37385401_37385556_0 . . . 353
13.1142 so_27826824_27826824_0 . . . 353
13.1143 so_47443172_47443409_0 . . . 353
13.1144 so_4061162_4061201_3 . . . 353
13.1145 so_30311798_30311848_0 . . . 353
13.1146 so_23505423_23505423_0 . . . 353
13.1147 so_23505423_23505744_1 . . . 353
13.1148 so_6474675_6474685_0 . . . 353
13.1149 so_47138819_47138978_0 . . . 353
13.1150 so_39514765_39515239_1 . . . 353
13.1151 so_7225962_7226010_4 . . . 353
13.1152 so_30428607_30428700_1 . . . 353
13.1153 so_9201497_9201526_0 . . . 353
13.1154 so_23039658_23039884_0 . . . 353
13.1155 so_28366444_28368013_5 . . . 353
13.1156 so_13403182_13403216_0 . . . 353
13.1157 so_33572768_33572822_0 . . . 353
13.1158 so_18201428_18202254_1 . . . 353
13.1159 so_34023464_34024297_0 . . . 353
13.1160 so_35943551_35943728_1 . . . 353
13.1161 so_19584395_19584496_0 . . . 353
13.1162 so_47307601_47308482_0 . . . 353
13.1163 so_36181895_36182442_0 . . . 353
13.1164 so_20841770_20841815_0 . . . 353
13.1165 so_37402250_37402319_0 . . . 353
13.1166 so_5683026_5683030_2 . . . 354
13.1167 so_40371000_40371240_1 . . . 354
13.1168 so_30681561_30681742_0 . . . 354
13.1169 so_23286828_23287233_0 . . . 354
13.1170 so_23595984_23595984_0 . . . 354
13.1171 so_15429265_15429303_0 . . . 354
13.1172 so_20888009_20888555_0 . . . 354
13.1173 so_13228028_13228049_2 . . . 354
13.1174 so_30681561_30681742_1 . . . 354
13.1175 so_28366444_28368013_6 . . . 354
13.1176 so_5683026_5683030_3 . . . 354
13.1177 so_23286828_23287233_1 . . . 354
13.1178 so_26470503_26471364_0 . . . 354
13.1179 so_12009865_12009901_0 . . . 354
13.1180 so_3088464_3088474_0 . . . 354
13.1181 so_27007831_27008033_1 . . . 354
13.1182 so_25291603_25291647_0 . . . 354
13.1183 so_11172389_11172459_4 . . . 354
13.1184 so_2241835_2241884_0 . . . 354
13.1185 so_40770829_40771065_0 . . . 354
13.1186 so_1516696_1516829_0 . . . 354
13.1187 so_32909109_32909355_0 . . . 354

13.1188 so_22186645_22186953_0 . . . 354
13.1189 so_23039658_23039884_1 . . . 354
13.1190 so_6493471_6493531_0 . . . 354
13.1191 so_25360835_25362435_4 . . . 354
13.1192 so_5150680_5150998_0 . . . 354
13.1193 so_8509712_8509712_1 . . . 354
13.1194 so_8509712_8509712_0 . . . 354
13.1195 so_44942904_44943252_1 . . . 354
13.1196 so_13267734_13267759_0 . . . 354
13.1197 so_32776445_32776545_1 . . . 355
13.1198 so_47307601_47308482_2 . . . 355
13.1199 so_27256619_27256786_0 . . . 355
13.1200 so_27256619_27256786_2 . . . 355
13.1201 so_29038703_29038870_1 . . . 355
13.1202 so_30636541_30637053_0 . . . 355
13.1203 so_45518910_45546008_1 . . . 355
13.1204 so_44429408_44437919_2 . . . 355
13.1205 so_44429408_44437919_0 . . . 355
13.1206 so_30409287_30409317_1 . . . 355
13.1207 so_30409287_30409313_0 . . . 355
13.1208 so_30409287_30409317_3 . . . 355
13.1209 so_22186645_22186953_1 . . . 355
13.1210 so_23039658_23039884_2 . . . 355
13.1211 so_8914986_8917084_0 . . . 355
13.1212 so_5194100_5194203_0 . . . 355
13.1213 so_35373384_35726062_0 . . . 355
13.1214 so_35373384_35375162_3 . . . 355
13.1215 so_35373384_35375162_1 . . . 355
13.1216 so_17210718_17210845_1 . . . 355
13.1217 so_17210718_17210845_3 . . . 355
13.1218 so_20433791_20433965_0 . . . 355
13.1219 so_6996600_6996740_0 . . . 355
13.1220 so_12724078_12724110_0 . . . 355
13.1221 so_15203541_15204239_0 . . . 355
13.1222 so_6894370_6901949_0 . . . 355
13.1223 so_21411098_21411211_1 . . . 355
13.1224 so_3906098_3906120_0 . . . 355
13.1225 so_4923270_4923639_0 . . . 355
13.1226 so_12365415_12365484_0 . . . 355
13.1227 so_22186645_22186953_3 . . . 355
13.1228 so_23039658_23039884_4 . . . 355
13.1229 so_34671555_34672121_0 . . . 355
13.1230 so_20583450_20583499_1 . . . 355
13.1231 so_20583450_20583499_3 . . . 355
13.1232 so_38542766_38542766_0 . . . 355
13.1233 so_6094732_6094772_0 . . . 355
13.1234 so_6094732_6094778_0 . . . 355
13.1235 so_1448596_1448673_0 . . . 355
13.1236 so_15244094_47483493_0 . . . 355
13.1237 so_3872260_3872345_0 . . . 355
13.1238 so_8754498_8754646_0 . . . 355

13.1239 so_24025336_24025412_0 . . . 355
13.1240 so_5270514_5270552_0 . . . 355
13.1241 so_23595984_23596045_0 . . . 355
13.1242 so_41506343_41507081_1 . . . 356
13.1243 so_12721584_12721584_2 . . . 356
13.1244 so_7358233_7375891_0 . . . 356
13.1245 so_24956922_24957263_0 . . . 356
13.1246 so_14368525_14368589_0 . . . 356
13.1247 so_16596864_16596909_1 . . . 356
13.1248 so_20440878_20441127_0 . . . 356
13.1249 so_29983869_29984044_0 . . . 356
13.1250 so_14505971_14505971_1 . . . 357
13.1251 so_41506343_41507081_0 . . . 357
13.1252 so_41506343_41507081_2 . . . 357
13.1253 so_29983869_29984358_3 . . . 357
13.1254 so_29983869_29984358_8 . . . 357
13.1255 so_29983869_29984358_4 . . . 357
13.1256 so_4385297_4385743_0 . . . 357
13.1257 so_4385297_4385297_7 . . . 357
13.1258 Built-in operators . . . 357
13.1259 so_27826824_27826995_0 . . . 357
13.1260 so_44515148_44515690_0 . . . 357
13.1261 so_33120453_33120598_3 . . . 357
13.1262 so_8914986_8915746_1 . . . 357
13.1263 so_4061162_4061262_0 . . . 357
13.1264 so_30804659_30805815_3 . . . 357
13.1265 so_17644816_17645573_0 . . . 357
13.1266 so_18482906_18483006_1 . . . 357
13.1267 so_9266648_9266914_0 . . . 357
13.1268 so_6804029_6804415_1 . . . 358
13.1269 so_3845229_3845256_2 . . . 358
13.1270 so_2033110_2033147_9 . . . 358
13.1271 so_45602368_45602856_1 . . . 358
13.1272 so_18889028_18889029_1 . . . 358
13.1273 so_9266648_9266648_0 . . . 358
13.1274 so_8914986_8915746_3 . . . 358
13.1275 so_17665110_17665272_0 . . . 358
13.1276 so_24489929_24489971_2 . . . 359
13.1277 so_21351450_21351616_0 . . . 359
13.1278 so_21351450_21351616_1 . . . 359
13.1279 so_5366080_5366094_1 . . . 359
13.1280 so_5366080_5366094_0 . . . 359
13.1281 so_31936418_31937090_1 . . . 359
13.1282 so_31936418_31937090_0 . . . 359
13.1283 so_25100855_25101393_3 . . . 359
13.1284 so_42421458_42422428_6 . . . 359
13.1285 so_42421458_42422428_5 . . . 359
13.1286 so_41637152_41637450_5 . . . 359
13.1287 so_24489929_24489971_1 . . . 359
13.1288 so_44679200_44679576_2 . . . 360
13.1289 so_41637152_41637450_6 . . . 360

13.1290 so_4691056_4691274_1 . . . 361

14 Templates **362**
14.1 so_36508532_36508532_0 . . . 362
14.2 so_36508532_36508532_1 . . . 362
14.3 so_45524824_45525034_2 . . . 362
14.4 so_30883473_30883739_2 . . . 362
14.5 so_12335762_12335796_6 . . . 362
14.6 so_12335762_12335796_8 . . . 362
14.7 so_12335762_12335796_4 . . . 362
14.8 so_12335762_12335796_7 . . . 362
14.9 so_12335762_12335796_3 . . . 362
14.10 so_12335762_12335796_5 . . . 362
14.11 so_28383035_28383177_0 . . . 362
14.12 so_46237554_46237554_3 . . . 362
14.13 so_30861592_30861592_0 . . . 362
14.14 so_37117796_37117796_0 . . . 362
14.15 so_18860895_42069703_1 . . . 362
14.16 so_19941203_19941429_2 . . . 362
14.17 so_19941203_19941429_0 . . . 362
14.18 so_45019980_45019980_1 . . . 362
14.19 so_7111957_7112168_1 . . . 362
14.20 so_42834902_42835144_0 . . . 362
14.21 so_20743582_20743681_1 . . . 362
14.22 so_28525031_28525031_0 . . . 362
14.23 so_33256589_33258249_19 . . . 362
14.24 so_33256589_33258249_16 . . . 362
14.25 so_33256589_33258249_9 . . . 362
14.26 so_33256589_33258249_17 . . . 362
14.27 so_33256589_33258249_10 . . . 362
14.28 so_33256589_33258249_8 . . . 362
14.29 so_33256589_33258249_6 . . . 362
14.30 so_33256589_33258249_18 . . . 362
14.31 so_33256589_33258249_7 . . . 362
14.32 so_33256589_33258249_20 . . . 362
14.33 so_33256589_33258249_21 . . . 362
14.34 so_21474111_21474111_0 . . . 362
14.35 so_23553723_23553808_1 . . . 362
14.36 so_28513114_28513114_2 . . . 362
14.37 so_28513114_28513114_3 . . . 362
14.38 so_28513114_28513114_1 . . . 362
14.39 so_12340105_12340271_7 . . . 362
14.40 so_12340105_12340271_4 . . . 362
14.41 so_12340105_12340271_6 . . . 362
14.42 so_12340105_12340271_5 . . . 362
14.43 so_12340105_12340271_8 . . . 362
14.44 so_12340105_12340271_3 . . . 362
14.45 so_41263504_41263504_0 . . . 362
14.46 so_41263504_41454584_1 . . . 362
14.47 so_46726390_46726390_0 . . . 362
14.48 so_31016784_31016833_2 . . . 362

14.49 so_46510358_46510398_1 . . . 362
14.50 so_29872879_29879700_1 . . . 362
14.51 so_32768034_32768225_1 . . . 362
14.52 so_32768034_32768223_0 . . . 362
14.53 so_46624005_47250175_1 . . . 362
14.54 so_46624005_47250175_5 . . . 362
14.55 so_28203734_28203734_0 . . . 362
14.56 so_28383035_28383177_1 . . . 362
14.57 so_40242505_40243839_1 . . . 362
14.58 so_3829771_3830216_0 . . . 362
14.59 so_34419603_34419745_1 . . . 362
14.60 so_6076946_6077022_0 . . . 362
14.61 so_30861592_30861924_1 . . . 363
14.62 so_6304259_6304259_0 . . . 363
14.63 so_4877705_4877705_1 . . . 363
14.64 so_3341055_3341055_0 . . . 363
14.65 so_3341055_3432119_1 . . . 363
14.66 so_19448165_19449007_0 . . . 363
14.67 so_994353_994428_1 . . . 363
14.68 so_994353_994428_3 . . . 363
14.69 so_37879642_37879642_3 . . . 363
14.70 so_37879642_37879642_2 . . . 363
14.71 so_37879642_37879642_0 . . . 363
14.72 so_28129777_28130423_1 . . . 363
14.73 so_5589381_5589381_1 . . . 363
14.74 so_23112409_23114645_1 . . . 363
14.75 so_36841401_36841449_7 . . . 363
14.76 so_19366615_24495429_0 . . . 363
14.77 so_28669262_28669262_0 . . . 363
14.78 so_34201090_34201090_0 . . . 363
14.79 so_31651326_31662268_6 . . . 363
14.80 so_20616627_20616771_0 . . . 363
14.81 so_5765924_5767006_0 . . . 363
14.82 so_21355115_21402060_0 . . . 363
14.83 so_3368129_3368191_4 . . . 363
14.84 so_41667547_41683684_0 . . . 363
14.85 so_46434964_46435142_1 . . . 363
14.86 so_46434964_46435266_0 . . . 363
14.87 so_6397657_6397680_0 . . . 363
14.88 so_22989281_22989829_0 . . . 363
14.89 so_24553297_24557347_0 . . . 363
14.90 so_6472081_6472196_2 . . . 363
14.91 so_19347671_19347715_0 . . . 363
14.92 so_26745973_26746325_0 . . . 363
14.93 so_9607089_9607089_1 . . . 363
14.94 so_42922961_42923438_0 . . . 363
14.95 so_18776844_18776892_2 . . . 363
14.96 so_11527415_11527505_0 . . . 363
14.97 Template parameters . . . 363
14.98 so_16431216_16439735_0 . . . 363
14.99 so_10268523_10268559_0 . . . 363

14.100 so_5777137_5777165_0 . . . 363
14.101 so_2023977_2024035_0 . . . 363
14.102 so_40440109_40440109_0 . . . 363
14.103 so_15369612_15369650_1 . . . 363
14.104 so_37074561_37075933_0 . . . 363
14.105 so_20980192_20981414_2 . . . 363
14.106 so_27256516_27256623_1 . . . 364
14.107 so_46610383_46610777_1 . . . 364
14.108 so_46624005_46624005_0 . . . 364
14.109 so_30139580_30141622_0 . . . 364
14.110 so_11336507_11373917_0 . . . 364
14.111 so_15175203_15175222_3 . . . 364
14.112 so_15175203_15175222_2 . . . 364
14.113 so_15175203_15175222_0 . . . 364
14.114 so_38154762_38154792_0 . . . 364
14.115 so_29559613_29560258_0 . . . 364
14.116 so_29559613_29560258_1 . . . 364
14.117 so_46153145_46153971_0 . . . 364
14.118 so_12717174_12717247_0 . . . 364
14.119 so_16214422_16214727_0 . . . 364
14.120 so_24092614_24093636_0 . . . 364
14.121 so_2033110_2033147_2 . . . 364
14.122 so_2033110_2033147_5 . . . 364
14.123 so_2033110_2033147_4 . . . 364
14.124 so_14691545_14691916_0 . . . 364
14.125 so_4259929_4335067_0 . . . 364
14.126 so_46027276_46027318_0 . . . 364
14.127 so_12983300_12983410_2 . . . 364
14.128 so_12983300_12983410_3 . . . 364
14.129 so_12983300_12983410_0 . . . 364
14.130 so_28379727_28379811_0 . . . 364
14.131 so_20066104_20066609_2 . . . 364
14.132 so_20066104_20066609_3 . . . 364
14.133 so_20066104_20066609_0 . . . 364
14.134 so_26290770_26290930_0 . . . 364
14.135 so_30930350_30930486_3 . . . 364
14.136 so_30930350_30930486_2 . . . 364
14.137 so_3261178_3261178_0 . . . 364
14.138 so_5687540_5687553_0 . . . 364
14.139 so_24790287_24790452_0 . . . 364
14.140 so_30126487_30126487_1 . . . 364
14.141 so_32173757_32173795_2 . . . 364
14.142 so_32173757_32173795_3 . . . 364
14.143 so_32173757_32173795_0 . . . 364
14.144 so_13543425_13543462_0 . . . 364
14.145 so_32235396_32235445_0 . . . 364
14.146 so_9679116_9679237_0 . . . 364
14.147 so_32173757_32173795_6 . . . 364
14.148 so_12983300_12983410_6 . . . 364
14.149 so_20066104_20066609_6 . . . 365
14.150 so_34029240_34029742_0 . . . 365

14.151 so_16693994_16693994_2 . . . 365
14.152 so_16693994_16693994_0 . . . 365
14.153 so_14266394_14266458_0 . . . 365
14.154 so_13543425_13543462_1 . . . 365
14.155 so_37666268_37666686_0 . . . 365
14.156 so_21841041_21841113_0 . . . 365
14.157 so_43528101_43528436_0 . . . 365
14.158 so_27810202_27810260_0 . . . 365
14.159 so_46994594_46994931_0 . . . 365
14.160 so_33027214_33027310_0 . . . 365
14.161 so_31493296_31493775_0 . . . 365
14.162 so_33146871_33147835_0 . . . 365
14.163 so_15977079_15977148_0 . . . 365
14.164 so_27909109_28764887_2 . . . 365
14.165 so_27909109_28764887_1 . . . 365
14.166 so_34561884_34675596_0 . . . 365
14.167 so_35150772_35155708_0 . . . 365
14.168 so_7452849_7452867_1 . . . 365
14.169 so_7452849_7452867_0 . . . 365
14.170 so_15369612_15369612_0 . . . 365
14.171 so_14812171_14812223_0 . . . 365
14.172 so_31624209_31625178_1 . . . 365
14.173 so_7881190_7881190_2 . . . 365
14.174 so_24114697_24114859_1 . . . 365
14.175 so_24114697_24114859_2 . . . 365
14.176 so_33146871_33147835_1 . . . 365
14.177 so_31678245_31695350_0 . . . 365
14.178 so_24362039_24362106_0 . . . 365
14.179 so_12067278_12068372_0 . . . 365
14.180 so_32414867_32414941_0 . . . 365
14.181 so_43993162_43994530_0 . . . 365
14.182 so_29776850_29777086_0 . . . 365
14.183 so_24017466_24026094_1 . . . 365
14.184 so_34940875_34941417_0 . . . 366
14.185 so_34940875_34941816_0 . . . 366
14.186 so_22951367_23298947_0 . . . 366
14.187 so_31697055_31697055_0 . . . 366
14.188 so_31697055_31700442_0 . . . 366
14.189 so_35701888_35701958_0 . . . 366
14.190 so_29219576_29220039_0 . . . 366
14.191 so_39137281_39137480_0 . . . 366
14.192 so_29941499_29941820_1 . . . 366
14.193 so_30344062_30352758_0 . . . 366
14.194 so_33690863_33691081_0 . . . 366
14.195 so_15977656_15977854_0 . . . 366
14.196 so_14768951_14769093_0 . . . 366
14.197 so_13753541_13753702_0 . . . 366
14.198 so_15222638_15222768_0 . . . 366
14.199 so_3472295_3473784_0 . . . 366
14.200 so_41625748_41626347_1 . . . 366
14.201 so_33661847_33661994_1 . . . 366

14.202 so_37955601_37955601_0 . . . 366
14.203 so_20981319_20981319_0 . . . 366
14.204 so_12046545_12046591_0 . . . 366
14.205 so_11684954_11685037_1 . . . 366
14.206 so_11684954_11685037_2 . . . 366
14.207 so_11684954_11685037_0 . . . 366
14.208 so_38629601_38629697_0 . . . 366
14.209 so_40704605_40705233_0 . . . 366
14.210 so_25893828_29915578_0 . . . 366
14.211 so_40053280_40053432_0 . . . 366
14.212 so_30679813_30679813_0 . . . 366
14.213 so_14197436_14197544_0 . . . 366
14.214 so_40812204_40812612_0 . . . 366
14.215 so_40812204_40812204_0 . . . 366
14.216 so_40812204_40812273_0 . . . 366
14.217 so_24017466_24026094_5 . . . 366
14.218 so_24017466_24022186_0 . . . 366
14.219 so_31954078_31956369_0 . . . 366
14.220 so_20464648_20476142_0 . . . 366
14.221 so_24582188_24582262_2 . . . 366
14.222 so_30292955_30469081_0 . . . 366
14.223 so_16011873_16012256_2 . . . 366
14.224 Names of template specializations . . . 367
14.225 so_40628584_40629271_0 . . . 367
14.226 so_40628584_40629271_5 . . . 367
14.227 so_7674695_7676073_1 . . . 367
14.228 so_24333686_24335198_4 . . . 367
14.229 so_24333686_24335198_5 . . . 367
14.230 so_18311496_18311496_3 . . . 367
14.231 so_3120951_3120974_0 . . . 367
14.232 so_28006660_28006861_1 . . . 367
14.233 so_4420828_4420839_0 . . . 367
14.234 so_5752215_5752223_0 . . . 367
14.235 so_18311496_18311496_4 . . . 367
14.236 so_29331315_29356768_0 . . . 367
14.237 so_10013623_10013642_0 . . . 367
14.238 so_28326857_28326857_0 . . . 367
14.239 so_10671406_10676746_0 . . . 367
14.240 so_33275718_33277159_0 . . . 367
14.241 so_33423702_33423740_0 . . . 367
14.242 so_33423702_33423740_2 . . . 367
14.243 so_610245_613132_3 . . . 367
14.244 so_28354108_28354108_0 . . . 367
14.245 so_28354108_28354108_2 . . . 367
14.246 so_19891489_19891924_0 . . . 367
14.247 so_19106622_19114118_0 . . . 367
14.248 so_28336366_28336366_0 . . . 367
14.249 so_28336366_28336366_1 . . . 367
14.250 so_41786026_41786026_0 . . . 367
14.251 so_27592885_27601859_7 . . . 368
14.252 so_27592885_27601859_5 . . . 368

14.253 so_44367859_44640490_0 . . . 368
14.254 so_12676190_12676256_0 . . . 368
14.255 so_10426428_10426708_4 . . . 368
14.256 so_26037290_26640789_0 . . . 368
14.257 so_12252536_12252562_0 . . . 368
14.258 so_43931734_43931925_0 . . . 368
14.259 so_22330845_22331715_0 . . . 368
14.260 so_1840253_1840318_0 . . . 368
14.261 so_31397066_31701116_0 . . . 368
14.262 so_2162510_2162593_0 . . . 368
14.263 so_26265618_26265976_0 . . . 368
14.264 so_6232294_6232526_0 . . . 368
14.265 so_30011740_30014205_2 . . . 368
14.266 so_27722358_27722358_9 . . . 368
14.267 so_4077110_4077164_0 . . . 368
14.268 so_3903336_3903336_1 . . . 368
14.269 so_3903336_3903336_0 . . . 368
14.270 so_5802077_5802122_0 . . . 368
14.271 so_7674695_7676073_0 . . . 368
14.272 so_33500299_33500369_0 . . . 368
14.273 so_14227721_14227746_0 . . . 368
14.274 so_3499101_3507906_0 . . . 368
14.275 so_3120951_3120974_1 . . . 368
14.276 so_32387428_32387467_0 . . . 368
14.277 so_8185870_8185972_0 . . . 368
14.278 so_1682844_1682959_0 . . . 368
14.279 so_25811206_25814596_7 . . . 368
14.280 so_25811206_25814596_2 . . . 368
14.281 so_30205514_30205781_2 . . . 368
14.282 so_30205514_30205845_1 . . . 368
14.283 so_30205514_30205781_1 . . . 368
14.284 so_3786360_3786481_0 . . . 368
14.285 so_15572415_15572442_0 . . . 368
14.286 so_31690932_31756207_0 . . . 368
14.287 so_20847637_20847776_0 . . . 368
14.288 so_18344580_27602255_5 . . . 368
14.289 so_18344580_18358008_0 . . . 368
14.290 so_18344580_18358008_2 . . . 368
14.291 so_18344580_18344580_3 . . . 368
14.292 so_34696351_34732843_4 . . . 368
14.293 so_28995673_28995825_0 . . . 368
14.294 so_28995673_28995825_4 . . . 368
14.295 so_24588207_24591796_11 . . . 368
14.296 so_24588207_24588297_0 . . . 368
14.297 so_24588207_24591796_15 . . . 368
14.298 so_5752215_5752223_1 . . . 368
14.299 so_24178083_24178216_0 . . . 368
14.300 so_29609955_29621984_0 . . . 368
14.301 so_46938763_46938811_1 . . . 368
14.302 so_6456636_6456886_0 . . . 368
14.303 so_6456636_6456636_0 . . . 368

14.304 so_6456636_6456886_2 . . . 368
14.305 so_6456636_6456886_1 . . . 368
14.306 so_39203315_39203893_0 . . . 368
14.307 so_26564049_26564114_2 . . . 368
14.308 so_31397066_31701116_1 . . . 368
14.309 so_31397066_31701116_2 . . . 368
14.310 so_9984612_9984742_1 . . . 368
14.311 so_9984612_9984742_0 . . . 368
14.312 so_24576931_24577107_2 . . . 368
14.313 so_24576931_24577107_0 . . . 368
14.314 so_27930448_28021419_9 . . . 368
14.315 so_27930448_28021419_10 . . . 368
14.316 so_27930448_28021419_11 . . . 368
14.317 so_27930448_28021419_8 . . . 368
14.318 so_27930448_28021419_3 . . . 368
14.319 so_8256636_8256685_0 . . . 368
14.320 so_25811206_25814596_4 . . . 368
14.321 so_25811206_25814596_9 . . . 368
14.322 so_20847637_20847776_1 . . . 368
14.323 so_46866912_46867139_0 . . . 368
14.324 so_46574697_46575224_0 . . . 368
14.325 so_24479553_24509400_0 . . . 369
14.326 Template arguments . . . 369
14.327 so_11060208_11060242_0 . . . 369
14.328 so_17378481_17379727_0 . . . 369
14.329 so_31345193_31345795_0 . . . 369
14.330 so_34714752_34716639_1 . . . 369
14.331 so_19121890_19122919_0 . . . 369
14.332 so_31033068_31034690_4 . . . 370
14.333 so_31488016_31488016_0 . . . 370
14.334 so_31488016_31488016_1 . . . 370
14.335 so_30245091_30245730_2 . . . 370
14.336 so_28102382_28102838_0 . . . 370
14.337 so_4638956_4638975_0 . . . 370
14.338 so_29865093_29865685_0 . . . 370
14.339 so_31404809_31404833_0 . . . 370
14.340 so_16322943_16692118_0 . . . 370
14.341 so_16720201_16720201_0 . . . 370
14.342 so_18596412_18600649_0 . . . 371
14.343 so_27909109_28764887_4 . . . 371
14.344 so_16527595_16527660_0 . . . 371
14.345 so_28887549_28887703_0 . . . 371
14.346 so_26553062_26553155_0 . . . 371
14.347 so_14270941_14271402_0 . . . 371
14.348 so_3377828_3378709_0 . . . 371
14.349 so_18871122_18871122_0 . . . 371
14.350 so_17378481_17379727_1 . . . 371
14.351 so_5939486_5941022_0 . . . 371
14.352 so_8336220_8336232_2 . . . 371
14.353 so_991062_991162_1 . . . 371
14.354 so_4569928_4569975_4 . . . 371

14.355 so_4569928_4569975_2 . . . 371
14.356 so_35695071_35695147_0 . . . 371
14.357 so_44548242_44549084_0 . . . 371
14.358 so_8203750_8203922_1 . . . 371
14.359 so_8203750_8203922_3 . . . 371
14.360 so_46552534_46552626_0 . . . 371
14.361 so_19958146_19958413_0 . . . 371
14.362 so_47473986_47478004_0 . . . 371
14.363 so_29112567_29112711_0 . . . 371
14.364 so_33005082_33005198_6 . . . 371
14.365 so_36033000_36033322_0 . . . 372
14.366 so_6804029_6804415_3 . . . 372
14.367 so_6804029_6804415_4 . . . 372
14.368 so_6804029_6804415_0 . . . 372
14.369 so_16575534_16576184_2 . . . 372
14.370 so_16575534_16576184_0 . . . 372
14.371 so_18498957_18499719_1 . . . 372
14.372 so_20377210_20377285_2 . . . 372
14.373 so_38960126_38961143_0 . . . 372
14.374 so_23479015_23479220_0 . . . 372
14.375 so_23479015_23479190_2 . . . 372
14.376 so_643763_643927_0 . . . 372
14.377 so_28614591_28614776_0 . . . 372
14.378 so_15175203_15175222_7 . . . 372
14.379 so_15175203_15175222_11 . . . 372
14.380 so_15175203_15175222_10 . . . 372
14.381 so_15175203_15175222_12 . . . 372
14.382 so_15175203_15175222_9 . . . 372
14.383 so_15175203_15175222_6 . . . 372
14.384 so_7392057_7392179_0 . . . 372
14.385 so_33301552_33301552_0 . . . 372
14.386 so_33301552_33302605_1 . . . 372
14.387 so_33301552_33302195_0 . . . 372
14.388 so_33301552_33301552_1 . . . 372
14.389 so_33301552_33301552_3 . . . 372
14.390 so_33301552_33302605_4 . . . 372
14.391 so_33872039_34017482_0 . . . 372
14.392 so_9183485_9183858_1 . . . 372
14.393 so_39777331_39777847_0 . . . 372
14.394 so_39777331_39777847_1 . . . 372
14.395 so_14192936_14192936_2 . . . 372
14.396 so_14192936_14192936_1 . . . 372
14.397 so_14192936_14192936_0 . . . 372
14.398 so_43528101_43528436_1 . . . 372
14.399 so_43528101_43528436_3 . . . 372
14.400 so_38154762_38155129_0 . . . 372
14.401 so_10180552_10181573_0 . . . 372
14.402 so_46153145_46153971_1 . . . 372
14.403 so_6742386_6745554_0 . . . 372
14.404 so_37303968_37304432_1 . . . 372
14.405 so_37303968_37304432_2 . . . 372

14.406 so_37303968_37304432_4 . . . 372
14.407 so_37303968_37304416_0 . . . 372
14.408 so_37303968_37304432_5 . . . 372
14.409 so_3845229_3845256_1 . . . 372
14.410 so_3845229_3845256_4 . . . 372
14.411 so_40088477_40089058_0 . . . 372
14.412 so_12632696_12633038_0 . . . 372
14.413 so_12632696_12633038_1 . . . 372
14.414 so_39440209_39441589_0 . . . 372
14.415 so_39440209_39441589_2 . . . 372
14.416 so_33692872_33692987_0 . . . 372
14.417 so_8629839_8629943_6 . . . 372
14.418 so_8629839_8629943_2 . . . 372
14.419 so_8629839_8629943_0 . . . 372
14.420 so_8629839_8629943_4 . . . 372
14.421 so_12717174_12717288_0 . . . 372
14.422 so_15175434_15175495_4 . . . 372
14.423 so_15175434_15175495_2 . . . 372
14.424 so_15175434_15175495_1 . . . 372
14.425 so_1826464_1826530_0 . . . 372
14.426 so_2033110_2033147_11 . . . 372
14.427 so_2033110_2033147_12 . . . 372
14.428 so_2033110_2033147_8 . . . 372
14.429 so_5605142_5612606_1 . . . 372
14.430 so_37557992_37558466_6 . . . 372
14.431 so_37557992_37558466_0 . . . 372
14.432 so_37557992_37558466_3 . . . 372
14.433 so_37557992_37558466_7 . . . 372
14.434 so_37557992_37558466_1 . . . 372
14.435 so_37557992_37558466_8 . . . 372
14.436 so_37557992_37558466_4 . . . 372
14.437 so_37557992_37558466_5 . . . 372
14.438 so_32788819_32789309_0 . . . 372
14.439 so_22284496_22286476_0 . . . 372
14.440 so_14691545_14691916_1 . . . 372
14.441 so_13606483_13607061_0 . . . 372
14.442 so_27909109_28764887_6 . . . 372
14.443 so_27909109_28764887_7 . . . 372
14.444 so_42070462_42070921_0 . . . 372
14.445 so_2548887_2549144_0 . . . 372
14.446 so_26124797_26125630_0 . . . 372
14.447 so_39776437_39778788_0 . . . 372
14.448 so_20662945_20664932_1 . . . 372
14.449 so_33957274_33958291_1 . . . 372
14.450 so_12983300_12983410_7 . . . 372
14.451 so_15867399_15867521_1 . . . 372
14.452 so_15867399_15867521_3 . . . 372
14.453 so_15410833_15411030_1 . . . 372
14.454 so_15410833_15411030_0 . . . 372
14.455 so_15410833_15411030_3 . . . 372
14.456 so_16711244_16712003_0 . . . 372

14.457 so_16711244_16712003_7 . . . 372
14.458 so_16711244_16712003_1 . . . 372
14.459 so_16711244_16712003_5 . . . 372
14.460 so_16711244_16712003_6 . . . 372
14.461 so_16711244_16712003_3 . . . 372
14.462 so_16711244_16712003_4 . . . 372
14.463 so_34561884_34675596_1 . . . 372
14.464 so_28662784_28728098_1 . . . 372
14.465 so_28662784_28662784_1 . . . 372
14.466 so_28662784_28728098_3 . . . 372
14.467 so_24442849_24442934_0 . . . 372
14.468 so_25595742_25598405_5 . . . 372
14.469 so_2183087_11518757_3 . . . 372
14.470 so_2183087_11518757_6 . . . 372
14.471 so_2183087_11518757_4 . . . 372
14.472 so_2183087_11518757_5 . . . 372
14.473 so_2183087_11518757_0 . . . 372
14.474 so_2183087_11518757_1 . . . 372
14.475 so_33838101_33838558_8 . . . 372
14.476 so_33838101_33838558_7 . . . 372
14.477 so_33838101_33838558_2 . . . 372
14.478 so_33838101_33838558_1 . . . 372
14.479 so_46328005_46345014_1 . . . 372
14.480 so_16004264_16004686_1 . . . 372
14.481 so_16004264_16004686_0 . . . 372
14.482 so_26582944_26583140_0 . . . 372
14.483 so_29001446_29001694_0 . . . 372
14.484 so_29001446_29001648_0 . . . 372
14.485 so_29001446_29001694_1 . . . 372
14.486 so_12024706_12024724_3 . . . 372
14.487 so_28809728_28810559_1 . . . 372
14.488 so_37326442_37326681_0 . . . 372
14.489 so_25946089_25946341_0 . . . 372
14.490 so_30604686_30604988_0 . . . 372
14.491 so_30604686_30604988_5 . . . 372
14.492 so_23793274_23793369_0 . . . 372
14.493 so_17354679_17354693_6 . . . 372
14.494 so_17354679_17354693_4 . . . 372
14.495 so_17354679_17354693_7 . . . 372
14.496 so_17354679_17354693_0 . . . 372
14.497 so_17354679_17354693_3 . . . 372
14.498 so_17354679_17354693_5 . . . 372
14.499 so_17354679_17354693_1 . . . 372
14.500 so_43996755_43997199_1 . . . 372
14.501 so_19564009_19565341_0 . . . 372
14.502 so_19564009_19566059_0 . . . 372
14.503 so_27395773_27396180_0 . . . 372
14.504 so_41085202_41086026_0 . . . 372
14.505 so_41085202_41086026_1 . . . 372
14.506 so_18352385_18352847_0 . . . 372
14.507 so_38173469_38230873_0 . . . 372

14.508 so_8232493_8232610_0 . . . 372
14.509 so_8232493_8232586_0 . . . 372
14.510 so_46769112_46769854_2 . . . 372
14.511 so_14417430_14417552_0 . . . 372
14.512 so_5592027_5592096_0 . . . 372
14.513 so_12910015_12910015_2 . . . 372
14.514 so_12910015_12910015_1 . . . 372
14.515 so_40787241_40787595_0 . . . 372
14.516 so_15885399_15885399_3 . . . 372
14.517 so_15885399_15885399_1 . . . 372
14.518 so_31003857_31004097_0 . . . 372
14.519 so_31003857_31004233_0 . . . 372
14.520 so_31003857_31004233_1 . . . 372
14.521 so_24877338_24877581_1 . . . 372
14.522 so_24877338_24877581_0 . . . 372
14.523 so_12717174_12717247_1 . . . 372
14.524 so_2033110_2033147_13 . . . 372
14.525 so_14242694_14242694_1 . . . 372
14.526 so_28809728_28810559_2 . . . 372
14.527 so_711779_711858_1 . . . 372
14.528 so_33443130_33443130_0 . . . 372
14.529 so_29984893_29985337_0 . . . 372
14.530 so_11222519_11222807_1 . . . 372
14.531 so_31448845_31448845_0 . . . 373
14.532 so_33301552_33301552_5 . . . 373
14.533 so_27410465_27411367_4 . . . 373
14.534 so_27410465_27411367_5 . . . 373
14.535 so_27410465_27411152_0 . . . 373
14.536 so_27410465_27411367_3 . . . 373
14.537 so_36033000_36033322_3 . . . 373
14.538 so_13562681_13563449_1 . . . 373
14.539 so_37290298_37290467_0 . . . 373
14.540 so_20377210_20377285_0 . . . 373
14.541 so_25532861_25533333_0 . . . 373
14.542 so_31448845_31449668_6 . . . 373
14.543 so_33301552_33301552_7 . . . 373
14.544 so_35313292_35313292_1 . . . 373
14.545 so_26559758_30424903_7 . . . 373
14.546 so_27625392_27625392_0 . . . 373
14.547 so_39440209_39441589_3 . . . 373
14.548 so_13606483_13607061_1 . . . 373
14.549 so_28717682_28718438_4 . . . 373
14.550 so_30461093_30461305_2 . . . 373
14.551 so_2548887_2549144_1 . . . 373
14.552 so_10031397_10031741_0 . . . 373
14.553 so_10031397_10031741_2 . . . 373
14.554 so_10031397_10031741_1 . . . 373
14.555 so_17559165_17559165_1 . . . 373
14.556 so_35153996_35154862_4 . . . 373
14.557 so_3542497_3542584_0 . . . 373
14.558 so_4027154_4027336_0 . . . 373

14.559 so_46328005_46345014_2 . . . 373
14.560 so_24345399_24346350_0 . . . 373
14.561 so_27121395_27121474_0 . . . 373
14.562 so_26290770_26290930_2 . . . 373
14.563 so_8098858_8100124_0 . . . 373
14.564 so_28482809_28483256_10 . . . 373
14.565 so_33107787_33108109_1 . . . 373
14.566 so_33107787_33108109_3 . . . 373
14.567 so_18352385_18353789_3 . . . 373
14.568 so_47278515_47278551_0 . . . 373
14.569 so_24826139_24826298_1 . . . 373
14.570 so_26309910_26310017_0 . . . 373
14.571 so_26309910_26309910_0 . . . 373
14.572 so_27806504_27806504_0 . . . 373
14.573 so_14290071_14290523_2 . . . 373
14.574 so_28184888_28184997_0 . . . 373
14.575 so_11060208_11060242_1 . . . 374
14.576 so_22281205_22281608_1 . . . 374
14.577 so_9435108_9435234_0 . . . 374
14.578 so_29219576_29220039_2 . . . 374
14.579 so_20291352_20291484_0 . . . 374
14.580 so_6626759_6627128_0 . . . 374
14.581 so_15651488_15651770_1 . . . 374
14.582 so_32104890_32426650_3 . . . 374
14.583 so_40063336_40075906_0 . . . 374
14.584 so_38494276_38494477_1 . . . 374
14.585 so_20253473_20276614_0 . . . 374
14.586 so_40201020_40201381_1 . . . 374
14.587 so_20356486_20356938_0 . . . 375
14.588 so_20356486_20356938_1 . . . 375
14.589 so_46328005_46345014_0 . . . 375
14.590 so_26060176_26060449_0 . . . 375
14.591 so_26060176_26060493_0 . . . 375
14.592 so_39813123_39813678_0 . . . 375
14.593 so_46309228_46311003_0 . . . 375
14.594 so_30292955_30469081_1 . . . 375
14.595 Type equivalence . . . 375
14.596 so_31448845_31449668_0 . . . 375
14.597 so_36711542_36712038_1 . . . 375
14.598 so_32788819_32789099_0 . . . 375
14.599 so_15501470_15502379_3 . . . 375
14.600 so_8667515_8667864_2 . . . 375
14.601 so_8667515_8667864_1 . . . 375
14.602 so_34817394_34819002_0 . . . 375
14.603 so_28662784_28662784_0 . . . 375
14.604 so_17689954_17690085_2 . . . 375
14.605 so_27410465_27411367_6 . . . 375
14.606 so_16693994_16694256_4 . . . 375
14.607 so_16693994_16694256_2 . . . 375
14.608 so_16693994_16694256_6 . . . 375
14.609 so_38311348_38311599_0 . . . 375

14.610 so_33337983_33338281_5 . . . 376
14.611 so_33337983_33338281_4 . . . 376
14.612 so_23263253_23271415_0 . . . 376
14.613 so_23263253_23275446_0 . . . 376
14.614 so_44294743_44413900_8 . . . 376
14.615 so_44294743_44413900_14 . . . 376
14.616 Template declarations . . . 376
14.617 so_26844443_26844613_0 . . . 377
14.618 so_7801228_7801370_0 . . . 377
14.619 so_6622452_6623089_4 . . . 377
14.620 so_28923515_28923515_0 . . . 377
14.621 so_7776448_7776448_0 . . . 377
14.622 so_20847637_20847637_1 . . . 377
14.623 so_29076190_29079664_0 . . . 377
14.624 so_31624209_31625178_0 . . . 377
14.625 so_42142800_42142800_0 . . . 377
14.626 so_39173519_39173519_0 . . . 377
14.627 so_9369845_9370714_0 . . . 377
14.628 so_32159000_32159000_0 . . . 377
14.629 so_28956250_28956250_0 . . . 377
14.630 so_29864305_29864445_0 . . . 377
14.631 so_7680274_7680681_0 . . . 378
14.632 so_7680274_7680681_1 . . . 378
14.633 so_16922023_16922023_1 . . . 378
14.634 so_27760896_27760976_0 . . . 378
14.635 so_34081179_34083702_0 . . . 379
14.636 so_5032683_5032794_0 . . . 379
14.637 so_22920665_22920748_0 . . . 379
14.638 so_32353894_32353894_0 . . . 379
14.639 so_19366615_19367075_1 . . . 379
14.640 so_27830761_27883353_1 . . . 379
14.641 so_22453111_22461577_0 . . . 379
14.642 so_31393530_31543319_0 . . . 379
14.643 so_31341917_31341955_0 . . . 379
14.644 so_40656857_40694056_0 . . . 379
14.645 so_20161535_20163170_0 . . . 379
14.646 so_3986765_3986778_0 . . . 379
14.647 so_41695963_41696136_2 . . . 379
14.648 so_879535_880323_0 . . . 379
14.649 so_11336507_11373917_4 . . . 379
14.650 so_24286523_24287590_4 . . . 380
14.651 so_15578298_15580072_6 . . . 380
14.652 so_9975291_9975452_3 . . . 380
14.653 so_598913_604022_1 . . . 380
14.654 so_25836109_25836942_3 . . . 380
14.655 so_23423966_23424604_6 . . . 380
14.656 so_23423966_23424604_13 . . . 380
14.657 so_15586526_15586567_4 . . . 380
14.658 so_28383035_28383177_2 . . . 380
14.659 so_14987326_14990231_3 . . . 380
14.660 so_15601057_15601166_3 . . . 380

14.661 so_33095439_33095469_5 . . . 380
14.662 so_2823485_2823485_1 . . . 380
14.663 so_40242505_40243839_2 . . . 380
14.664 so_16635822_16636025_4 . . . 380
14.665 so_25047109_25047700_2 . . . 380
14.666 so_19959637_19959637_0 . . . 380
14.667 so_37117796_37118038_0 . . . 380
14.668 so_11752307_11752401_0 . . . 380
14.669 so_26858076_26858409_4 . . . 380
14.670 so_17209102_17209430_0 . . . 380
14.671 so_937744_937762_0 . . . 380
14.672 so_15497004_15497032_3 . . . 380
14.673 so_15208198_15208571_6 . . . 380
14.674 so_15208198_15208300_1 . . . 380
14.675 so_10278144_10278594_2 . . . 380
14.676 so_16764285_16764306_3 . . . 380
14.677 so_8203750_8203860_2 . . . 380
14.678 so_26799551_26800526_0 . . . 380
14.679 so_2778352_2778387_0 . . . 380
14.680 so_757270_757364_0 . . . 380
14.681 so_20513692_20513752_0 . . . 380
14.682 so_7628231_7628340_0 . . . 380
14.683 so_7628231_7628259_0 . . . 380
14.684 so_2886193_2886218_0 . . . 380
14.685 so_25353085_25353919_1 . . . 380
14.686 so_24924603_24924660_0 . . . 381
14.687 so_16922023_16922023_0 . . . 381
14.688 so_38352691_38352905_0 . . . 381
14.689 so_2786946_2786963_0 . . . 381
14.690 so_25866147_25866471_0 . . . 381
14.691 so_6358882_6359067_0 . . . 381
14.692 so_25320366_25320622_0 . . . 381
14.693 so_9701525_9701570_0 . . . 381
14.694 so_26553803_26557917_0 . . . 381
14.695 so_28850608_28850776_0 . . . 381
14.696 so_2861839_2861859_0 . . . 381
14.697 so_425457_425615_0 . . . 381
14.698 so_13002368_13002620_0 . . . 381
14.699 so_43808848_43809035_1 . . . 381
14.700 so_1447554_1447560_0 . . . 381
14.701 so_6230448_6230473_0 . . . 381
14.702 so_25047109_25048348_2 . . . 381
14.703 so_25047109_25048348_3 . . . 381
14.704 so_7972263_7972434_0 . . . 381
14.705 so_46049209_46049461_1 . . . 381
14.706 so_7972263_7972434_1 . . . 381
14.707 so_7972263_7972434_2 . . . 382
14.708 so_14191989_14192162_0 . . . 382
14.709 so_26681920_26682069_0 . . . 382
14.710 so_18991910_18993037_0 . . . 382
14.711 so_28887549_28887703_2 . . . 382

14.712 so_12048221_12048366_0 . . . 382
14.713 so_40380203_40381315_0 . . . 382
14.714 so_7972263_7972434_3 . . . 382
14.715 so_24647926_24648091_0 . . . 382
14.716 so_13503877_13504228_0 . . . 382
14.717 so_24582188_24582262_1 . . . 382
14.718 so_24582188_24582262_13 . . . 382
14.719 so_24582188_24582262_11 . . . 382
14.720 so_24582188_24582262_3 . . . 382
14.721 so_24582188_24582262_0 . . . 382
14.722 so_24582188_24582262_6 . . . 382
14.723 so_24582188_24582262_4 . . . 382
14.724 so_31388928_31389185_0 . . . 382
14.725 so_31388928_31389185_1 . . . 382
14.726 so_38146184_38146416_0 . . . 382
14.727 so_43849659_43849659_0 . . . 382
14.728 so_31489351_31489351_0 . . . 382
14.729 so_31489351_31489351_1 . . . 382
14.730 so_20423080_20423615_0 . . . 382
14.731 so_37996053_37996340_0 . . . 382
14.732 so_36984193_36985664_0 . . . 382
14.733 so_38763282_38763282_0 . . . 382
14.734 so_16011873_16012256_0 . . . 382
14.735 so_16011873_16012256_4 . . . 382
14.736 so_16011873_16012256_3 . . . 382
14.737 so_16011873_16012256_1 . . . 382
14.738 so_12515616_12515637_0 . . . 382
14.739 so_12515616_12515975_0 . . . 382
14.740 so_40380203_40380203_0 . . . 382
14.741 so_7694201_7716499_1 . . . 382
14.742 so_7340909_7342744_0 . . . 382
14.743 so_23401376_23711944_0 . . . 382
14.744 so_14575347_14575450_0 . . . 382
14.745 so_14575347_14575450_1 . . . 382
14.746 so_33385979_33386355_1 . . . 382
14.747 so_18382002_18668877_0 . . . 382
14.748 so_34940875_34940875_0 . . . 383
14.749 so_14270941_14271107_0 . . . 383
14.750 so_47034088_47236038_1 . . . 383
14.751 so_13503877_13504228_1 . . . 383
14.752 so_30819547_30819548_5 . . . 383
14.753 so_38146184_38146424_1 . . . 383
14.754 so_38146184_38146424_2 . . . 383
14.755 so_13025535_13025809_0 . . . 383
14.756 so_10426428_10426708_1 . . . 384
14.757 so_21602957_21605768_0 . . . 384
14.758 so_21602957_21605768_1 . . . 384
14.759 so_15064535_15066266_1 . . . 384
14.760 so_19459283_19459283_4 . . . 384
14.761 so_19459283_19459283_1 . . . 384
14.762 so_19459283_19459283_5 . . . 384

14.763 so_19459283_19459283_2 . . . 384
14.764 so_19459283_19459283_0 . . . 384
14.765 so_19459283_19459283_3 . . . 384
14.766 so_29945432_29945584_0 . . . 384
14.767 so_38562865_38579114_2 . . . 384
14.768 so_38562865_38579114_3 . . . 384
14.769 so_38562865_38579114_0 . . . 384
14.770 so_38562865_38579114_4 . . . 384
14.771 so_38562865_38579114_1 . . . 384
14.772 so_36644566_36645467_2 . . . 384
14.773 so_43732122_43732722_4 . . . 384
14.774 so_32964090_32964293_0 . . . 384
14.775 so_11296389_12610403_6 . . . 384
14.776 so_12573816_35891188_1 . . . 384
14.777 so_42692050_42692128_0 . . . 384
14.778 so_37933007_37933007_1 . . . 384
14.779 so_13521546_13521546_3 . . . 384
14.780 so_13521546_13521546_1 . . . 384
14.781 so_13521546_13537141_0 . . . 384
14.782 so_13521546_13521546_4 . . . 384
14.783 so_13521546_13521546_2 . . . 384
14.784 so_13521546_13521546_0 . . . 384
14.785 so_7111957_7112168_4 . . . 384
14.786 so_31624209_31624209_0 . . . 384
14.787 so_15208198_15208348_0 . . . 384
14.788 so_35997875_35999952_2 . . . 384
14.789 so_43645582_43645824_2 . . . 384
14.790 so_29871138_29871138_0 . . . 384
14.791 so_18099166_18100355_0 . . . 384
14.792 so_8514880_8514893_5 . . . 384
14.793 so_8514880_8514893_7 . . . 384
14.794 so_8514880_8514893_3 . . . 384
14.795 so_8514880_8514893_6 . . . 384
14.796 so_8514880_8514893_4 . . . 384
14.797 so_34899524_34899663_1 . . . 384
14.798 so_3431735_3431811_2 . . . 384
14.799 so_1599171_1599194_0 . . . 384
14.800 so_27719934_27720191_0 . . . 385
14.801 so_5118371_5118371_0 . . . 385
14.802 so_46103102_46103645_0 . . . 385
14.803 so_46103102_46103102_0 . . . 385
14.804 so_8203750_8203860_3 . . . 385
14.805 so_16567212_16567212_0 . . . 386
14.806 so_3736350_3736981_0 . . . 386
14.807 so_11041274_11046918_0 . . . 386
14.808 so_27754396_27754474_2 . . . 386
14.809 so_18212329_18212526_0 . . . 386
14.810 so_33563427_33564333_0 . . . 386
14.811 so_13055447_13057570_1 . . . 386
14.812 so_24501388_24503971_0 . . . 386
14.813 so_25362727_25362727_0 . . . 386

14.814 so_28923515_28923630_0 . . . 386
14.815 so_17580273_17580390_4 . . . 386
14.816 so_31755029_31755465_0 . . . 386
14.817 so_3368129_3368191_5 . . . 386
14.818 so_28734686_28734686_2 . . . 386
14.819 so_17132753_17154766_1 . . . 386
14.820 so_41143524_41143524_1 . . . 386
14.821 so_6304259_6304259_1 . . . 386
14.822 so_27678342_27678509_0 . . . 386
14.823 so_9582034_9582687_1 . . . 386
14.824 so_9582034_9585856_0 . . . 386
14.825 so_13923684_13925256_0 . . . 386
14.826 so_40656857_40694056_1 . . . 386
14.827 so_16880885_16880917_0 . . . 386
14.828 so_16765179_16765329_0 . . . 386
14.829 so_27678342_27678509_2 . . . 386
14.830 so_47344285_47344491_0 . . . 386
14.831 so_19861596_19861596_0 . . . 387
14.832 so_29550788_29550808_0 . . . 387
14.833 so_40656857_40656857_0 . . . 387
14.834 so_40685032_40691509_0 . . . 387
14.835 so_33099306_33099716_1 . . . 387
14.836 so_17200312_17200312_0 . . . 387
14.837 so_31393530_31543319_1 . . . 387
14.838 so_32104890_32426650_1 . . . 387
14.839 so_25362727_25362875_0 . . . 387
14.840 so_12858839_30991097_2 . . . 387
14.841 so_4543720_4543783_0 . . . 387
14.842 so_36293088_36293088_0 . . . 387
14.843 so_36293088_36293522_0 . . . 387
14.844 so_7776448_7776468_0 . . . 387
14.845 so_40616223_40616839_0 . . . 387
14.846 so_5188434_5188545_0 . . . 387
14.847 so_14359550_14359550_0 . . . 387
14.848 so_35486160_35486160_0 . . . 387
14.849 so_47252509_47252509_0 . . . 387
14.850 so_41143524_41143524_2 . . . 387
14.851 so_32104890_32426650_2 . . . 387
14.852 so_32104890_32104890_0 . . . 387
14.853 so_24656212_24656420_0 . . . 387
14.854 so_24656212_24656420_1 . . . 387
14.855 so_24656212_24656420_3 . . . 387
14.856 so_22486386_22486607_0 . . . 387
14.857 so_30927196_30927196_0 . . . 387
14.858 so_37578880_37579096_0 . . . 387
14.859 so_27678342_27678509_3 . . . 387
14.860 so_41238470_41238470_0 . . . 387
14.861 so_18856239_18856908_0 . . . 387
14.862 so_27742271_27742637_2 . . . 387
14.863 so_27742271_27742637_0 . . . 387
14.864 so_27742271_27742637_3 . . . 387

14.865 so_8866881_10471851_0 . . . 387
14.866 so_42142800_42142800_1 . . . 387
14.867 so_42142800_42142800_2 . . . 387
14.868 so_42142800_42143510_0 . . . 387
14.869 so_9582034_9582687_3 . . . 387
14.870 so_9582034_9582687_2 . . . 387
14.871 so_19902601_19906025_0 . . . 387
14.872 so_12446229_12498050_0 . . . 387
14.873 so_17200312_17200312_1 . . . 388
14.874 so_24961899_24962102_0 . . . 388
14.875 so_24961899_24962102_1 . . . 388
14.876 so_24961899_24962102_3 . . . 388
14.877 so_24961899_24962102_2 . . . 388
14.878 so_22828120_22828304_1 . . . 388
14.879 so_22828120_22828304_0 . . . 388
14.880 so_35486160_35486160_2 . . . 388
14.881 so_20987347_20987751_0 . . . 388
14.882 so_20268114_20273619_1 . . . 388
14.883 so_28469604_28470089_0 . . . 388
14.884 so_25483764_25485314_0 . . . 388
14.885 so_11132573_11132910_1 . . . 388
14.886 so_11132573_11132910_0 . . . 388
14.887 so_31497357_31595016_0 . . . 388
14.888 so_28519704_28519731_1 . . . 388
14.889 so_28519704_28519731_0 . . . 388
14.890 so_34714752_35725616_0 . . . 388
14.891 so_34714752_35725616_2 . . . 388
14.892 so_34714752_35725616_3 . . . 388
14.893 so_26317798_26318330_2 . . . 388
14.894 so_9582034_9582687_0 . . . 388
14.895 so_40892048_40892048_1 . . . 388
14.896 so_32320293_32320293_0 . . . 388
14.897 so_32320293_34536235_0 . . . 388
14.898 so_8850759_8850759_0 . . . 388
14.899 so_35875829_35877464_0 . . . 388
14.900 so_22281205_22281608_2 . . . 388
14.901 so_24961899_24962102_4 . . . 388
14.902 so_30676839_32572451_0 . . . 388
14.903 so_20268114_20273619_2 . . . 388
14.904 so_27687389_27688405_0 . . . 388
14.905 so_25483764_25485314_1 . . . 388
14.906 so_23362391_23430522_0 . . . 388
14.907 so_32104890_32426650_4 . . . 388
14.908 so_34714752_34714752_1 . . . 388
14.909 so_34714752_35725616_1 . . . 388
14.910 so_32320293_32320293_1 . . . 388
14.911 so_32104890_32426650_5 . . . 388
14.912 so_32320293_32320293_2 . . . 388
14.913 so_32320293_34536235_1 . . . 388
14.914 so_11615532_11616312_0 . . . 388
14.915 so_34552056_34552056_1 . . . 388

14.916 so_34552056_34716046_0 . . . 388
14.917 so_34552056_34716046_1 . . . 388
14.918 so_32104890_32426650_6 . . . 388
14.919 so_26081078_26081078_0 . . . 388
14.920 so_46054228_46054902_3 . . . 389
14.921 so_31394260_31735126_2 . . . 389
14.922 so_31475016_31475554_4 . . . 389
14.923 so_42244250_42246348_1 . . . 389
14.924 so_42244250_42246348_4 . . . 389
14.925 so_17005985_17008568_5 . . . 389
14.926 so_17005985_17008568_4 . . . 389
14.927 so_17005985_17008568_6 . . . 389
14.928 so_6757691_6757830_0 . . . 389
14.929 so_30676839_32572451_4 . . . 389
14.930 so_31391172_31391456_3 . . . 389
14.931 so_13629552_13629552_0 . . . 389
14.932 so_34552056_34716046_5 . . . 389
14.933 so_29961100_29961422_7 . . . 389
14.934 so_31497357_31735144_0 . . . 389
14.935 so_28519704_28519704_1 . . . 389
14.936 so_28519704_28519704_2 . . . 389
14.937 so_28519704_30252004_1 . . . 389
14.938 so_28519704_28519704_0 . . . 389
14.939 so_46309228_46311003_1 . . . 389
14.940 so_23804312_23806297_9 . . . 389
14.941 so_23804312_23806297_10 . . . 389
14.942 so_40892048_40892048_4 . . . 389
14.943 so_40892048_40894796_1 . . . 389
14.944 so_40892048_40892048_3 . . . 389
14.945 so_40892048_40894796_0 . . . 389
14.946 so_40892048_40894796_2 . . . 389
14.947 so_40892048_40892048_2 . . . 389
14.948 so_13923684_13925256_1 . . . 389
14.949 so_27320993_27321150_0 . . . 389
14.950 so_27320993_27321150_1 . . . 389
14.951 so_23544066_23545475_0 . . . 389
14.952 so_26987216_26988110_0 . . . 389
14.953 so_20793345_20793676_1 . . . 389
14.954 so_16880885_16880917_1 . . . 389
14.955 so_26440493_26440687_1 . . . 389
14.956 so_28193521_29545974_0 . . . 389
14.957 so_41143524_41143524_0 . . . 389
14.958 so_5206080_5206117_0 . . . 389
14.959 so_23750750_23750750_0 . . . 389
14.960 so_31624209_31625178_2 . . . 389
14.961 so_16765179_16765329_1 . . . 389
14.962 so_15374841_15378899_0 . . . 389
14.963 so_6138439_6138483_0 . . . 389
14.964 so_19861596_19863548_0 . . . 390
14.965 so_2097811_2098016_0 . . . 390
14.966 so_25709653_25709691_0 . . . 390

14.967 so_1875296_1875377_1 . . . 390
14.968 so_3052579_3052604_0 . . . 390
14.969 so_28669262_28669321_1 . . . 390
14.970 so_28669262_28669321_0 . . . 390
14.971 so_28669262_28669321_2 . . . 390
14.972 so_28669262_28669321_4 . . . 390
14.973 so_16865225_16865452_0 . . . 391
14.974 so_10374180_10374588_1 . . . 391
14.975 so_2174300_2175718_0 . . . 391
14.976 so_45769950_45800815_6 . . . 391
14.977 so_13629552_13630331_0 . . . 391
14.978 so_45769950_45800815_7 . . . 391
14.979 so_10374180_10374588_2 . . . 391
14.980 so_40157489_40159402_0 . . . 391
14.981 so_31648458_31649056_5 . . . 391
14.982 so_34303507_34303507_3 . . . 391
14.983 so_34303507_34303507_6 . . . 391
14.984 so_22923628_22923628_0 . . . 391
14.985 so_36722470_36725226_0 . . . 391
14.986 so_28203981_28204207_1 . . . 391
14.987 so_40266806_40267527_1 . . . 391
14.988 so_28203981_28204207_0 . . . 392
14.989 so_28203981_28204207_3 . . . 392
14.990 so_30917253_30917254_0 . . . 392
14.991 so_40266806_40267527_2 . . . 392
14.992 so_14420908_14420908_11 . . . 392
14.993 so_21602957_21605768_3 . . . 392
14.994 so_21602957_21605768_4 . . . 392
14.995 so_10374180_10374588_5 . . . 392
14.996 so_10374180_10374588_9 . . . 392
14.997 so_10374180_10374588_7 . . . 392
14.998 so_10374180_10374588_4 . . . 392
14.999 so_10374180_10374588_6 . . . 392
14.1000 so_10374180_10374588_8 . . . 392
14.1001 so_31497357_31735144_1 . . . 392
14.1002 so_20999417_24539642_5 . . . 392
14.1003 so_20999417_24539642_1 . . . 392
14.1004 so_20999417_24539642_4 . . . 392
14.1005 so_20999417_24539642_3 . . . 392
14.1006 so_20999417_24539642_2 . . . 392
14.1007 so_19503847_19503928_0 . . . 393
14.1008 so_14005020_14005020_0 . . . 393
14.1009 so_10374180_10374588_10 . . . 393
14.1010 so_9728693_9728764_3 . . . 393
14.1011 so_42957146_42957361_0 . . . 393
14.1012 so_28308599_28309212_1 . . . 393
14.1013 so_1180325_1193395_0 . . . 393
14.1014 so_17332320_17333188_2 . . . 393
14.1015 so_15497004_15497032_5 . . . 393
14.1016 so_20999417_24539642_6 . . . 393
14.1017 so_16764285_16764306_5 . . . 393

14.1018 so_40892048_40894796_3 . . . 393
14.1019 so_19365571_19406528_0 . . . 393
14.1020 so_17005985_17008568_1 . . . 393
14.1021 so_31051189_31051322_1 . . . 393
14.1022 so_42957146_42957361_1 . . . 393
14.1023 so_1180325_1193395_1 . . . 393
14.1024 so_17332320_17333188_3 . . . 393
14.1025 so_32373979_32373979_8 . . . 393
14.1026 so_32373979_32375678_2 . . . 393
14.1027 so_40892048_40894796_4 . . . 393
14.1028 so_19365571_19406528_1 . . . 393
14.1029 so_17332320_17333188_4 . . . 393
14.1030 so_40892048_40894796_5 . . . 393
14.1031 so_36051645_36072815_0 . . . 394
14.1032 so_32768034_32768225_0 . . . 395
14.1033 so_38555749_40203287_1 . . . 395
14.1034 so_29945432_29945584_1 . . . 395
14.1035 so_24219357_24219411_2 . . . 395
14.1036 so_24219357_24219411_1 . . . 395
14.1037 so_30242897_30243528_0 . . . 395
14.1038 so_38311348_38311599_1 . . . 395
14.1039 so_19438407_19438466_1 . . . 395
14.1040 so_30650073_30654483_2 . . . 395
14.1041 so_22951367_23298947_1 . . . 395
14.1042 so_17392621_17393256_0 . . . 395
14.1043 so_38555749_40203287_2 . . . 395
14.1044 so_22281205_22281608_0 . . . 395
14.1045 so_8775202_8775296_0 . . . 395
14.1046 so_31225888_35571258_0 . . . 395
14.1047 so_32723988_32724096_1 . . . 395
14.1048 so_21395786_21397053_0 . . . 395
14.1049 so_43619075_43636262_1 . . . 395
14.1050 so_47473986_47478004_2 . . . 395
14.1051 so_20851788_20851978_0 . . . 395
14.1052 so_41642359_42542260_0 . . . 395
14.1053 so_29867841_29868348_0 . . . 395
14.1054 so_33010352_33011069_0 . . . 395
14.1055 so_28967819_28968169_1 . . . 395
14.1056 so_28967819_28968169_2 . . . 395
14.1057 so_30650073_30654483_1 . . . 395
14.1058 so_29851098_29851257_1 . . . 395
14.1059 so_26574926_26574988_0 . . . 395
14.1060 so_10321856_10322052_0 . . . 395
14.1061 Name resolution . . . 396
14.1062 so_24547961_24548280_3 . . . 396
14.1063 so_24547961_24548280_0 . . . 396
14.1064 so_24547961_24548280_1 . . . 396
14.1065 so_27592885_27601859_1 . . . 396
14.1066 so_438192_438717_1 . . . 396
14.1067 so_34941638_34941719_0 . . . 396
14.1068 so_2023977_2024035_1 . . . 396

14.1069 so_34846386_34846460_0 . . . 396
14.1070 so_25156901_25157251_0 . . . 396
14.1071 so_14002565_14002610_0 . . . 396
14.1072 so_5704498_5704908_0 . . . 396
14.1073 so_1301380_1301390_0 . . . 396
14.1074 so_18344580_18344580_1 . . . 396
14.1075 so_18344580_27602255_1 . . . 396
14.1076 so_610245_613132_2 . . . 396
14.1077 so_31610809_31610964_0 . . . 396
14.1078 so_47312983_47312983_1 . . . 396
14.1079 so_4347730_4347986_0 . . . 396
14.1080 so_27592885_27601859_2 . . . 396
14.1081 so_26564049_26564114_0 . . . 396
14.1082 so_32377212_32378723_0 . . . 396
14.1083 so_27713666_27714133_0 . . . 396
14.1084 so_40756681_40756829_0 . . . 396
14.1085 so_6642721_6644062_1 . . . 396
14.1086 so_30554119_30554202_1 . . . 396
14.1087 so_40523513_40523853_1 . . . 396
14.1088 so_30205514_30205781_0 . . . 396
14.1089 so_21210969_21211176_1 . . . 396
14.1090 so_18344580_27602255_2 . . . 396
14.1091 so_7722276_7722407_0 . . . 396
14.1092 so_37080796_37080868_3 . . . 396
14.1093 so_37080796_37080868_2 . . . 396
14.1094 so_27720222_27720501_0 . . . 396
14.1095 so_30554119_30554202_2 . . . 396
14.1096 so_40523513_40523853_0 . . . 396
14.1097 so_4420828_4420839_2 . . . 396
14.1098 so_6642721_6644062_0 . . . 397
14.1099 so_27592885_27601859_3 . . . 397
14.1100 so_39186871_39186871_0 . . . 397
14.1101 so_47177212_47199026_0 . . . 397
14.1102 so_18344580_27602255_3 . . . 397
14.1103 so_30232485_30232943_0 . . . 398
14.1104 so_30232485_30234634_0 . . . 398
14.1105 so_42706880_42707066_0 . . . 398
14.1106 so_25822376_25822398_3 . . . 398
14.1107 so_25822376_25822398_1 . . . 398
14.1108 so_43749115_43762955_0 . . . 398
14.1109 so_43749115_43766415_1 . . . 398
14.1110 so_13255826_13256601_0 . . . 398
14.1111 so_29363532_29364222_0 . . . 398
14.1112 so_45489631_45489752_2 . . . 398
14.1113 so_34149992_34149992_2 . . . 398
14.1114 so_34149992_34149992_10 . . . 398
14.1115 so_46624005_47250175_7 . . . 398
14.1116 so_10993720_10997184_1 . . . 398
14.1117 so_30673043_30673381_0 . . . 398
14.1118 so_38834149_38834461_0 . . . 398
14.1119 so_40841777_40993650_3 . . . 398

14.1120 so_40841777_40993650_1 . . . 398
14.1121 so_30078818_30079285_0 . . . 398
14.1122 so_33413960_33414336_0 . . . 398
14.1123 so_27867949_27868077_0 . . . 398
14.1124 so_10323980_10324234_0 . . . 398
14.1125 so_11918814_11918879_0 . . . 398
14.1126 so_29468724_29470096_0 . . . 398
14.1127 so_40344285_40345460_0 . . . 398
14.1128 so_31681856_31682099_4 . . . 398
14.1129 so_24964760_24964971_1 . . . 398
14.1130 so_17598699_17599026_0 . . . 398
14.1131 so_27738971_27739161_0 . . . 398
14.1132 so_14648434_14649812_3 . . . 398
14.1133 so_40702274_40703439_0 . . . 398
14.1134 so_40702274_40703439_2 . . . 398
14.1135 so_40702274_40703439_1 . . . 398
14.1136 so_29463711_29463883_0 . . . 398
14.1137 so_32918009_32919319_0 . . . 398
14.1138 so_31345193_31345795_1 . . . 398
14.1139 so_14637356_14637534_0 . . . 398
14.1140 so_27021240_27021316_3 . . . 398
14.1141 so_38304847_38316977_0 . . . 398
14.1142 so_23263253_23275446_4 . . . 398
14.1143 so_32920488_32920640_0 . . . 398
14.1144 so_44059557_44064262_0 . . . 398
14.1145 so_37731401_38060482_2 . . . 398
14.1146 so_37731401_38060482_1 . . . 398
14.1147 so_37731401_38060482_0 . . . 398
14.1148 so_47432221_47437965_0 . . . 398
14.1149 so_32729353_32729398_0 . . . 398
14.1150 so_42913556_42913797_0 . . . 398
14.1151 so_40842044_40842044_0 . . . 398
14.1152 so_24933592_24934001_0 . . . 398
14.1153 so_27343811_27343880_0 . . . 398
14.1154 so_13257103_13257244_0 . . . 398
14.1155 so_30232485_30234634_1 . . . 398
14.1156 so_3749196_3749196_0 . . . 398
14.1157 so_9303843_9306682_0 . . . 398
14.1158 so_17478621_17478621_1 . . . 398
14.1159 so_4870739_4870775_0 . . . 398
14.1160 so_47136153_47136624_0 . . . 398
14.1161 so_14749461_14750180_0 . . . 398
14.1162 so_1239908_1254095_0 . . . 398
14.1163 so_39218851_39220703_0 . . . 398
14.1164 so_21582032_21582150_0 . . . 398
14.1165 so_3829040_3829146_2 . . . 398
14.1166 so_30232485_30234634_2 . . . 399
14.1167 so_9303843_9306682_1 . . . 399
14.1168 so_10993720_10995476_1 . . . 399
14.1169 so_30011740_30014205_3 . . . 399
14.1170 so_45518910_45546008_0 . . . 399

14.1171 so_29541875_29542086_0 . . . 399
14.1172 so_33863069_33878015_0 . . . 399
14.1173 so_25890042_25890408_0 . . . 399
14.1174 so_37563745_37564502_0 . . . 399
14.1175 so_21582032_21582150_1 . . . 399
14.1176 so_30437431_30437435_7 . . . 399
14.1177 so_28792179_28792389_1 . . . 399
14.1178 so_25051734_25052211_0 . . . 399
14.1179 so_16127770_16128324_0 . . . 399
14.1180 so_16127770_16128324_1 . . . 399
14.1181 so_27271941_27272134_0 . . . 399
14.1182 so_32037146_32037250_2 . . . 399
14.1183 so_12810630_12810755_0 . . . 399
14.1184 so_12810630_12810755_1 . . . 399
14.1185 so_30891207_30891261_0 . . . 399
14.1186 so_14291438_14291520_0 . . . 399
14.1187 so_27722358_27732038_12 . . . 399
14.1188 so_19013071_19013921_0 . . . 399
14.1189 so_25038479_25038543_1 . . . 399
14.1190 so_14227721_14227763_1 . . . 399
14.1191 so_16938480_16938744_0 . . . 399
14.1192 so_31177609_32401503_1 . . . 399
14.1193 so_11423380_11423718_1 . . . 399
14.1194 so_46963794_46964265_0 . . . 399
14.1195 so_17287778_17288193_1 . . . 399
14.1196 so_24307512_24307629_0 . . . 399
14.1197 so_26558507_26682739_16 . . . 399
14.1198 so_26558507_26682739_12 . . . 399
14.1199 so_26558507_26682739_15 . . . 399
14.1200 so_26558507_26682739_17 . . . 399
14.1201 so_26558507_26682739_13 . . . 399
14.1202 so_26558507_26682739_14 . . . 399
14.1203 so_37926349_37926439_0 . . . 399
14.1204 so_10339012_10339411_0 . . . 399
14.1205 so_24479553_24509400_1 . . . 399
14.1206 so_33746865_33747605_0 . . . 399
14.1207 so_8887864_8887980_3 . . . 399
14.1208 so_24216335_24216384_0 . . . 399
14.1209 so_28344878_28345846_0 . . . 399
14.1210 so_19886094_19886144_1 . . . 399
14.1211 so_34246443_34246498_0 . . . 399
14.1212 so_14882730_14882925_0 . . . 399
14.1213 so_19224685_19224908_0 . . . 399
14.1214 so_6170867_6170891_0 . . . 399
14.1215 so_40563997_40564182_0 . . . 399
14.1216 so_17687459_17687568_0 . . . 399
14.1217 so_8100492_8454314_0 . . . 399
14.1218 so_8031823_8031859_0 . . . 399
14.1219 so_20808993_20809467_1 . . . 399
14.1220 so_26195283_26195424_0 . . . 399
14.1221 so_26591103_26591234_0 . . . 399

14.1222 so_30891207_30891261_1 . . . 400
14.1223 so_19831327_19831496_0 . . . 400
14.1224 so_8574059_8574272_0 . . . 400
14.1225 so_28344878_28345846_1 . . . 400
14.1226 so_8100492_8454314_1 . . . 400
14.1227 so_5149119_5149468_7 . . . 400
14.1228 so_24307512_24307629_2 . . . 400
14.1229 so_8887864_8887980_4 . . . 400
14.1230 so_7025054_7025401_0 . . . 400
14.1231 so_14782568_14782795_0 . . . 400
14.1232 so_21118605_21118927_0 . . . 400
14.1233 so_28344878_28345846_6 . . . 400
14.1234 so_8100492_8454314_2 . . . 400
14.1235 so_17287778_17288193_2 . . . 400
14.1236 so_28344878_28345846_4 . . . 400
14.1237 so_8100492_8123034_0 . . . 400
14.1238 so_28029722_28029821_0 . . . 401
14.1239 so_486508_486597_0 . . . 401
14.1240 so_4573952_4573978_0 . . . 401
14.1241 so_11336507_11373917_2 . . . 401
14.1242 so_35744299_35744530_0 . . . 401
14.1243 so_38063271_38063957_0 . . . 401
14.1244 so_46992006_46993212_3 . . . 402
14.1245 so_24125841_24126011_2 . . . 402
14.1246 so_24125841_24126011_1 . . . 402
14.1247 so_28029722_28029821_1 . . . 402
14.1248 so_28029722_28029722_0 . . . 402
14.1249 so_31457627_31457917_0 . . . 402
14.1250 so_4296924_4297052_3 . . . 402
14.1251 so_18475953_18489952_2 . . . 402
14.1252 so_27402292_27404546_3 . . . 402
14.1253 so_851584_851760_2 . . . 402
14.1254 so_2767329_2767420_0 . . . 402
14.1255 so_29574616_29575182_2 . . . 402
14.1256 so_33907864_33908092_2 . . . 402
14.1257 so_40674779_40685194_2 . . . 402
14.1258 so_25188353_25188640_1 . . . 402
14.1259 so_18218475_18218475_4 . . . 402
14.1260 so_18218475_18218475_2 . . . 402
14.1261 so_27178483_27179338_1 . . . 402
14.1262 so_33721762_33721762_2 . . . 402
14.1263 so_45489631_45489752_3 . . . 402
14.1264 so_32235765_32604022_4 . . . 402
14.1265 so_13660958_13661081_0 . . . 402
14.1266 so_30078818_30081493_1 . . . 402
14.1267 so_45518910_45546008_2 . . . 402
14.1268 so_27067433_27067700_7 . . . 402
14.1269 so_28187170_28188355_3 . . . 402
14.1270 so_18048144_18048144_0 . . . 402
14.1271 so_30066698_30066794_1 . . . 402
14.1272 so_3951209_3951339_1 . . . 402

14.1273 so_3951209_3952874_2 . . . 402
14.1274 so_8501294_8501421_1 . . . 402
14.1275 so_6456636_6456886_4 . . . 402
14.1276 so_17772960_17773200_0 . . . 403
14.1277 so_4870739_4870775_1 . . . 403
14.1278 so_21636966_21676769_0 . . . 403
14.1279 so_27336766_27336844_0 . . . 403
14.1280 so_32399844_32399993_0 . . . 403
14.1281 so_9782625_9782772_0 . . . 403
14.1282 so_36647336_36647755_0 . . . 403
14.1283 so_44118934_44125608_0 . . . 403
14.1284 so_44416954_44417452_0 . . . 403
14.1285 so_32665178_32665178_0 . . . 403
14.1286 so_18867889_18869108_0 . . . 403
14.1287 so_31177609_32401503_2 . . . 403
14.1288 so_1567730_1567867_0 . . . 403
14.1289 so_33739289_33739488_0 . . . 403
14.1290 so_4643074_4643093_0 . . . 403
14.1291 so_38572374_38575475_2 . . . 403
14.1292 so_1239908_1254095_1 . . . 403
14.1293 so_22479641_22481091_2 . . . 403
14.1294 so_22479641_22481091_1 . . . 403
14.1295 so_36545819_36548889_2 . . . 403
14.1296 so_24307512_24308040_0 . . . 403
14.1297 so_24307512_24307629_3 . . . 403
14.1298 so_14534316_14534441_0 . . . 403
14.1299 so_257288_261262_2 . . . 403
14.1300 so_37504020_37504163_0 . . . 403
14.1301 so_8191908_8192219_0 . . . 403
14.1302 so_19338452_19339440_0 . . . 403
14.1303 so_33746865_33747605_2 . . . 403
14.1304 so_27888730_27888957_0 . . . 403
14.1305 so_43781269_43781574_4 . . . 403
14.1306 so_3829040_3829146_3 . . . 403
14.1307 so_11405_11425_5 . . . 403
14.1308 so_11405_11425_3 . . . 403
14.1309 so_11405_11425_4 . . . 403
14.1310 so_24623071_24623071_0 . . . 403
14.1311 so_32621370_32621475_1 . . . 403
14.1312 so_32621370_32621475_0 . . . 403
14.1313 so_29030988_29031231_0 . . . 403
14.1314 so_39228033_39229147_1 . . . 403
14.1315 so_28792179_28792389_0 . . . 403
14.1316 so_1643035_1643190_0 . . . 403
14.1317 so_32797599_32797990_1 . . . 403
14.1318 so_32797599_32797990_0 . . . 403
14.1319 so_7076169_7076326_0 . . . 403
14.1320 so_32377212_32378723_2 . . . 403
14.1321 so_14227721_14227746_1 . . . 403
14.1322 so_39363961_39374845_1 . . . 403
14.1323 so_17287778_17288193_0 . . . 403

14.1324 so_30205514_30205845_2 . . . 403
14.1325 so_1704353_1704441_0 . . . 403
14.1326 so_21903741_21904148_0 . . . 403
14.1327 so_18344580_18358008_3 . . . 403
14.1328 so_28995673_28995825_2 . . . 403
14.1329 so_28995673_28995825_1 . . . 403
14.1330 so_28995673_28995825_3 . . . 403
14.1331 so_24588207_24591796_13 . . . 403
14.1332 so_24588207_24591796_12 . . . 403
14.1333 so_29609955_29621984_1 . . . 403
14.1334 so_31659851_31660281_2 . . . 403
14.1335 so_18218475_18218475_5 . . . 404
14.1336 so_10426428_10426708_2 . . . 404
14.1337 so_32377212_32378723_3 . . . 404
14.1338 so_15064535_15066266_2 . . . 404
14.1339 so_27722358_27732455_2 . . . 404
14.1340 so_27722358_27732455_1 . . . 404
14.1341 so_27722358_27722358_11 . . . 404
14.1342 so_27722358_27732455_4 . . . 404
14.1343 so_17642624_17648572_2 . . . 404
14.1344 so_33746865_33747605_3 . . . 404
14.1345 so_18344580_18358008_1 . . . 404
14.1346 so_16887608_16915869_4 . . . 404
14.1347 so_27663038_27666854_6 . . . 404
14.1348 so_34696351_34732843_1 . . . 404
14.1349 so_17713124_17713124_0 . . . 404
14.1350 so_24588207_24591796_16 . . . 404
14.1351 so_24588207_24591796_18 . . . 404
14.1352 so_35997875_35999952_3 . . . 404
14.1353 so_27720222_27720501_2 . . . 404
14.1354 so_27720222_27720222_1 . . . 404
14.1355 so_18218475_18218475_7 . . . 405
14.1356 so_18218475_18218475_9 . . . 405
14.1357 so_18218475_18218475_8 . . . 405
14.1358 so_30011740_30014205_4 . . . 405
14.1359 so_27722358_27722358_13 . . . 405
14.1360 so_27722358_27722358_14 . . . 405
14.1361 so_27722358_27722358_12 . . . 405
14.1362 so_14188535_14309329_1 . . . 405
14.1363 so_14188535_14309329_0 . . . 405
14.1364 so_38572374_38575475_0 . . . 405
14.1365 so_13993798_14005063_0 . . . 405
14.1366 so_37905409_37905447_1 . . . 405
14.1367 so_13993798_13995360_0 . . . 406
14.1368 so_17478621_17479105_0 . . . 406
14.1369 so_21636966_21676769_1 . . . 406
14.1370 so_27688528_27688528_10 . . . 406
14.1371 so_27688528_27688528_9 . . . 406
14.1372 so_27688528_27688528_8 . . . 406
14.1373 so_27688528_27688528_6 . . . 406
14.1374 so_27688528_27688528_7 . . . 406

14.1375 so_27688528_27688528_5 . . . 406
14.1376 so_32377212_32378723_1 . . . 406
14.1377 so_17478621_17478621_0 . . . 406
14.1378 so_27713666_27714133_4 . . . 406
14.1379 so_27713666_27713666_10 . . . 406
14.1380 so_27713666_27714133_3 . . . 406
14.1381 so_27713666_27713666_11 . . . 406
14.1382 so_27722358_27732038_16 . . . 406
14.1383 so_3831158_3831176_0 . . . 406
14.1384 so_16887608_16888165_0 . . . 406
14.1385 so_24588207_24591796_3 . . . 406
14.1386 so_28967819_28969108_4 . . . 406
14.1387 so_28967819_28969108_1 . . . 406
14.1388 so_47312983_47312983_0 . . . 406
14.1389 so_27720222_27720222_6 . . . 406
14.1390 so_27720222_27720222_5 . . . 406
14.1391 so_27713222_27713597_1 . . . 406
14.1392 so_27722358_27732038_10 . . . 406
14.1393 so_37563745_37564502_1 . . . 406
14.1394 so_37563745_37564518_1 . . . 406
14.1395 so_16887608_16888165_2 . . . 406
14.1396 so_34696351_34732843_3 . . . 406
14.1397 so_34696351_34732843_0 . . . 406
14.1398 so_24588207_24591796_5 . . . 406
14.1399 so_27720222_27720222_8 . . . 406
14.1400 so_33721762_33721762_3 . . . 406
14.1401 so_27722358_27732038_8 . . . 406
14.1402 so_17516314_17516314_1 . . . 406
14.1403 so_24588207_24588297_1 . . . 406
14.1404 so_24588207_24591796_4 . . . 406
14.1405 so_38082969_38082969_0 . . . 406
14.1406 so_7674695_7676073_2 . . . 406
14.1407 so_27663038_27666854_2 . . . 406
14.1408 so_27663038_27666854_3 . . . 406
14.1409 so_27663038_27666854_4 . . . 406
14.1410 so_24588207_24591796_7 . . . 406
14.1411 so_24588207_24591796_10 . . . 406
14.1412 so_24588207_24591796_9 . . . 406
14.1413 so_27720222_27720222_13 . . . 406
14.1414 so_27720222_27720222_14 . . . 406
14.1415 so_27720222_27720222_15 . . . 406
14.1416 so_27720222_27720222_11 . . . 406
14.1417 so_27713222_27713222_3 . . . 407
14.1418 so_27713222_27713597_3 . . . 407
14.1419 so_27713666_27714133_5 . . . 407
14.1420 so_27713666_27713666_13 . . . 407
14.1421 so_27722358_27722358_3 . . . 407
14.1422 so_30360119_30360231_0 . . . 407
14.1423 so_43286980_43287800_0 . . . 407
14.1424 so_44294743_44413900_17 . . . 407
14.1425 so_27722358_27732455_0 . . . 407

14.1426 so_27722358_27722358_18 . . . 407
14.1427 so_14188535_14309329_2 . . . 407
14.1428 so_39218851_39220554_0 . . . 407
14.1429 so_34696351_34732843_2 . . . 407
14.1430 so_24588207_24591796_19 . . . 407
14.1431 so_27713222_27713597_2 . . . 407
14.1432 so_27713222_27713597_4 . . . 408
14.1433 so_27713222_27713222_5 . . . 408
14.1434 so_27722358_27722358_5 . . . 408
14.1435 so_16887608_16888165_1 . . . 408
14.1436 so_16887608_16915869_2 . . . 408
14.1437 so_8400791_8400791_2 . . . 408
14.1438 so_43286980_43287800_1 . . . 408
14.1439 so_27722358_27732455_3 . . . 408
14.1440 so_24158887_24159201_10 . . . 408
14.1441 so_24158887_24159201_11 . . . 408
14.1442 so_3793151_3793151_0 . . . 408
14.1443 so_30380787_30380981_0 . . . 408
14.1444 so_3793151_3793151_1 . . . 408
14.1445 so_32235765_32604022_6 . . . 409
14.1446 so_34154770_34157327_1 . . . 409
14.1447 so_12813351_12815815_0 . . . 409
14.1448 so_7369314_7370311_0 . . . 409
14.1449 so_12530174_12531411_0 . . . 409
14.1450 so_24500891_24501091_0 . . . 409
14.1451 so_8766576_8766619_0 . . . 409
14.1452 so_34929388_34929686_0 . . . 409
14.1453 so_18206923_18206923_6 . . . 409
14.1454 so_18206923_18206923_7 . . . 409
14.1455 so_18206923_18206923_5 . . . 409
14.1456 so_25188353_25188640_2 . . . 409
14.1457 so_25188353_25188640_4 . . . 409
14.1458 so_25188353_25188640_3 . . . 409
14.1459 so_35818887_35820028_0 . . . 409
14.1460 so_32235765_32235765_2 . . . 409
14.1461 so_32235765_32235765_0 . . . 409
14.1462 so_32235765_32597412_0 . . . 409
14.1463 so_32235765_32235765_1 . . . 409
14.1464 so_15975739_15976200_2 . . . 409
14.1465 so_15975739_15976200_0 . . . 409
14.1466 so_15975739_15976200_1 . . . 409
14.1467 so_27711306_27711626_0 . . . 409
14.1468 so_17478621_17478621_3 . . . 409
14.1469 so_17478621_17478621_4 . . . 409
14.1470 so_16548379_16548379_0 . . . 409
14.1471 so_18493826_18516246_0 . . . 409
14.1472 so_16544974_16546151_0 . . . 409
14.1473 so_19728109_19728661_3 . . . 409
14.1474 so_19728109_19728661_2 . . . 409
14.1475 so_19728109_19728661_1 . . . 409
14.1476 so_29831654_29831841_0 . . . 409

14.1477 so_13223278_13223306_2 . . . 409
14.1478 so_13223278_13223306_1 . . . 409
14.1479 so_13223278_13223306_0 . . . 409
14.1480 so_30066698_30066794_2 . . . 409
14.1481 so_30829648_30830594_1 . . . 409
14.1482 so_30103639_30104794_0 . . . 409
14.1483 so_36104071_36104426_3 . . . 409
14.1484 so_36104071_36104426_1 . . . 409
14.1485 so_36104071_36104426_2 . . . 409
14.1486 so_18598862_18599231_0 . . . 409
14.1487 so_26790855_26791206_2 . . . 409
14.1488 so_3810269_3810269_0 . . . 409
14.1489 so_28127126_28127759_0 . . . 409
14.1490 so_39059511_39086337_0 . . . 409
14.1491 so_32422636_32423546_0 . . . 409
14.1492 so_7369314_7370311_1 . . . 409
14.1493 so_23030403_23030403_0 . . . 409
14.1494 so_36997351_36999343_0 . . . 409
14.1495 so_33434606_33434606_0 . . . 409
14.1496 so_29889604_29890206_0 . . . 409
14.1497 so_1110418_1110485_2 . . . 409
14.1498 so_3810269_3810269_1 . . . 409
14.1499 so_15232758_15236647_2 . . . 409
14.1500 so_3810269_3817419_0 . . . 409
14.1501 so_3810269_3810269_2 . . . 409
14.1502 so_3810269_3817419_1 . . . 409
14.1503 so_17478621_17479105_2 . . . 409
14.1504 so_18174368_25736581_0 . . . 409
14.1505 so_37816186_37818272_1 . . . 409
14.1506 so_37742749_37742749_0 . . . 409
14.1507 so_37742749_37743365_3 . . . 409
14.1508 so_18174368_18174368_0 . . . 410
14.1509 so_18174368_25736581_1 . . . 410
14.1510 so_3483932_3484031_0 . . . 410
14.1511 so_32555618_32559162_0 . . . 410
14.1512 so_26790855_26791206_1 . . . 410
14.1513 so_25188353_25188640_5 . . . 410
14.1514 so_804217_804686_0 . . . 410
14.1515 so_17478621_17479105_1 . . . 410
14.1516 so_3810281_37500074_0 . . . 410
14.1517 so_28904035_28904035_1 . . . 410
14.1518 so_39925842_39926298_1 . . . 410
14.1519 so_25188353_25189247_0 . . . 410
14.1520 so_1197340_1197490_0 . . . 410
14.1521 so_28127126_28127759_1 . . . 410
14.1522 so_15258525_15258991_4 . . . 410
14.1523 so_35303841_35304967_1 . . . 410
14.1524 so_43793824_43793983_0 . . . 410
14.1525 so_23030403_23031288_0 . . . 410
14.1526 so_23030403_23031288_5 . . . 410
14.1527 so_23030403_23031288_6 . . . 410

14.1528 so_23030403_23031288_7 . . . 410
14.1529 so_18174368_25736581_2 . . . 410
14.1530 so_43764311_43768437_0 . . . 410
14.1531 so_34204483_34204707_0 . . . 410
14.1532 so_32555618_32559162_1 . . . 410
14.1533 so_26790855_26790855_1 . . . 410
14.1534 so_26790855_26790855_2 . . . 410
14.1535 so_14597463_14597463_6 . . . 410
14.1536 so_14597463_14597463_12 . . . 410
14.1537 so_39925842_39926298_0 . . . 410
14.1538 so_25188353_25188640_8 . . . 410
14.1539 so_25188353_25188640_6 . . . 410
14.1540 so_25188353_25188521_2 . . . 410
14.1541 so_25188353_25188521_0 . . . 410
14.1542 so_25188353_25188521_1 . . . 410
14.1543 so_25188353_25188640_7 . . . 410
14.1544 so_25188353_25188521_3 . . . 410
14.1545 so_25788127_31652888_1 . . . 410
14.1546 so_804217_804686_1 . . . 410
14.1547 so_27178483_27179338_2 . . . 410
14.1548 so_15258525_15258525_0 . . . 410
14.1549 so_154469_156834_4 . . . 410
14.1550 so_154469_156834_3 . . . 410
14.1551 so_154469_156834_5 . . . 410
14.1552 so_11453737_11454817_1 . . . 410
14.1553 so_25775874_25888855_0 . . . 410
14.1554 so_25775874_25888855_1 . . . 410
14.1555 so_7255281_7256320_2 . . . 410
14.1556 so_7255281_7256320_3 . . . 410
14.1557 so_7255281_7256320_4 . . . 410
14.1558 so_7255281_7256320_1 . . . 410
14.1559 so_33673614_36058677_1 . . . 410
14.1560 so_33673614_36058677_2 . . . 410
14.1561 so_33673614_36058677_3 . . . 410
14.1562 so_33673614_36058677_0 . . . 410
14.1563 so_26821777_26821911_0 . . . 410
14.1564 so_1396458_1402325_0 . . . 410
14.1565 so_27067433_27067700_0 . . . 410
14.1566 so_27067433_27067700_2 . . . 410
14.1567 so_27067433_27067700_1 . . . 410
14.1568 so_24533091_24533452_1 . . . 410
14.1569 so_24533091_24533452_0 . . . 410
14.1570 so_24533091_24533452_2 . . . 410
14.1571 so_13316727_13316839_2 . . . 410
14.1572 so_13316727_13316839_1 . . . 410
14.1573 so_13316727_13316839_0 . . . 410
14.1574 so_7369314_7370311_2 . . . 410
14.1575 so_17661308_17662745_2 . . . 410
14.1576 so_17661308_17662745_4 . . . 410
14.1577 so_17661308_17662745_5 . . . 410
14.1578 so_17661308_17662745_1 . . . 410

14.1579 so_17661308_17662745_3 . . . 410
14.1580 so_35737741_35737741_0 . . . 410
14.1581 so_43764311_43768437_3 . . . 410
14.1582 so_43764311_43768437_4 . . . 410
14.1583 so_43764311_43768437_2 . . . 410
14.1584 so_43764311_43768437_1 . . . 410
14.1585 so_14595992_14596065_3 . . . 410
14.1586 so_28904035_28904035_4 . . . 410
14.1587 so_28904035_28904035_3 . . . 410
14.1588 so_28904035_28904035_2 . . . 410
14.1589 so_14787818_14787818_1 . . . 410
14.1590 so_14787818_14787818_2 . . . 410
14.1591 so_23809774_23814084_1 . . . 410
14.1592 so_23809774_23814084_0 . . . 410
14.1593 so_23809774_23814084_3 . . . 410
14.1594 so_23809774_23814084_2 . . . 410
14.1595 so_21384604_21385079_2 . . . 410
14.1596 so_21384604_21385079_1 . . . 410
14.1597 so_21384604_21385079_0 . . . 410
14.1598 so_26835352_26835352_1 . . . 410
14.1599 so_26835352_26835352_0 . . . 410
14.1600 so_8501294_8501421_4 . . . 410
14.1601 so_8501294_8501421_3 . . . 410
14.1602 so_25295345_25295745_3 . . . 410
14.1603 so_25295345_25295745_2 . . . 410
14.1604 so_25295345_25295745_0 . . . 410
14.1605 so_25295345_25295745_1 . . . 410
14.1606 so_18598862_18598942_2 . . . 410
14.1607 so_18598862_18598942_1 . . . 410
14.1608 so_46242144_46242688_1 . . . 410
14.1609 so_32624557_32624557_0 . . . 410
14.1610 so_8379002_8380830_1 . . . 410
14.1611 so_4274386_4279182_3 . . . 410
14.1612 so_32624557_32624874_0 . . . 410
14.1613 so_32624557_32624874_1 . . . 410
14.1614 Template instantiation and specialization . . . 411
14.1615 so_28734686_28734686_0 . . . 411
14.1616 so_28469604_28469604_0 . . . 411
14.1617 so_12676190_12676190_0 . . . 411
14.1618 so_5500057_30949326_1 . . . 411
14.1619 so_26673081_26673205_0 . . . 411
14.1620 so_32788663_32788961_0 . . . 411
14.1621 so_47314166_47314443_0 . . . 411
14.1622 so_46526988_46526988_0 . . . 411
14.1623 so_4464300_4464300_0 . . . 411
14.1624 so_20817378_20818023_0 . . . 411
14.1625 so_46624005_47250175_4 . . . 412
14.1626 so_46624005_47250175_6 . . . 412
14.1627 so_28734686_28734686_1 . . . 412
14.1628 so_28734686_28734686_3 . . . 412
14.1629 so_28469604_28470316_0 . . . 412

14.1630 so_13521546_13521546_5 . . . 412
14.1631 so_41915587_41916709_4 . . . 412
14.1632 so_41915587_41916709_2 . . . 412
14.1633 so_5912689_5912745_1 . . . 412
14.1634 so_5912689_5912745_0 . . . 412
14.1635 so_24333686_24333686_0 . . . 412
14.1636 so_26305368_26305550_1 . . . 412
14.1637 so_26305368_26305550_0 . . . 412
14.1638 so_21534435_21534435_0 . . . 412
14.1639 so_47544299_47544299_2 . . . 412
14.1640 so_47544299_47544299_0 . . . 412
14.1641 so_37345946_37346109_0 . . . 412
14.1642 so_17689954_17690085_0 . . . 412
14.1643 so_17689954_17690085_1 . . . 412
14.1644 so_42574656_42574862_0 . . . 412
14.1645 so_7787301_7787405_0 . . . 412
14.1646 so_36608839_36608988_1 . . . 412
14.1647 so_36608839_36608988_0 . . . 412
14.1648 so_24420340_24420535_0 . . . 412
14.1649 so_28814973_28814973_0 . . . 412
14.1650 so_28814973_28815144_0 . . . 412
14.1651 so_7046647_7047049_2 . . . 412
14.1652 so_44044267_44044683_0 . . . 412
14.1653 so_45081096_45081406_1 . . . 412
14.1654 so_45081096_45081406_2 . . . 412
14.1655 so_17478621_17478621_2 . . . 412
14.1656 so_21166762_21166957_0 . . . 412
14.1657 so_7515123_7516359_0 . . . 412
14.1658 so_21803992_21803992_0 . . . 412
14.1659 so_26323841_26323863_0 . . . 412
14.1660 so_19341360_19341402_1 . . . 412
14.1661 so_25397011_25397133_0 . . . 412
14.1662 so_26672776_26672859_1 . . . 412
14.1663 so_25195429_25195622_0 . . . 412
14.1664 so_8379002_8380830_0 . . . 412
14.1665 so_40377100_40377459_1 . . . 412
14.1666 so_47219453_47219453_0 . . . 412
14.1667 so_18570632_18571345_0 . . . 412
14.1668 so_22055403_22055550_0 . . . 412
14.1669 so_28469604_28469604_1 . . . 412
14.1670 so_2892087_2892133_0 . . . 412
14.1671 so_33504651_33511131_1 . . . 412
14.1672 so_35461561_35463025_0 . . . 412
14.1673 so_29263302_29263302_0 . . . 412
14.1674 so_5072554_5072740_0 . . . 412
14.1675 so_17132114_17132624_1 . . . 412
14.1676 so_6688118_6688626_0 . . . 412
14.1677 so_25925551_26005482_2 . . . 412
14.1678 so_4295661_4296545_0 . . . 412
14.1679 so_25351809_25351809_0 . . . 412
14.1680 so_39530837_39536077_0 . . . 412

14.1681 so_28772593_28772593_0 . . . 412
14.1682 so_23679525_23679903_0 . . . 412
14.1683 so_27678981_27679361_1 . . . 412
14.1684 so_28870175_28870175_0 . . . 412
14.1685 so_37816186_37818272_0 . . . 412
14.1686 so_30415930_30416005_0 . . . 412
14.1687 so_19964575_19964793_1 . . . 412
14.1688 so_46395724_46395975_1 . . . 412
14.1689 so_3408469_8686069_0 . . . 412
14.1690 so_28715062_28715062_0 . . . 412
14.1691 so_17528836_17528912_1 . . . 412
14.1692 so_26422189_26422452_1 . . . 412
14.1693 so_34885025_34885535_0 . . . 412
14.1694 so_25644109_25644427_0 . . . 412
14.1695 so_26673081_26673081_0 . . . 412
14.1696 so_2299523_2299660_0 . . . 412
14.1697 so_28578061_28579464_0 . . . 412
14.1698 so_17466466_17467393_1 . . . 412
14.1699 so_41689767_41690334_0 . . . 412
14.1700 so_46341681_46935513_2 . . . 412
14.1701 so_29697469_29697590_0 . . . 412
14.1702 so_28643625_28643625_0 . . . 412
14.1703 so_7730874_7731106_0 . . . 412
14.1704 so_30232485_30233665_0 . . . 413
14.1705 so_27760896_27760976_2 . . . 413
14.1706 so_15891106_15891160_2 . . . 413
14.1707 so_7515123_7516359_1 . . . 413
14.1708 so_47196863_47230580_0 . . . 413
14.1709 so_38884954_38885680_0 . . . 413
14.1710 so_19341360_19341623_0 . . . 413
14.1711 so_15234572_15234667_2 . . . 413
14.1712 so_25525688_25526090_0 . . . 413
14.1713 so_25525688_25527229_0 . . . 413
14.1714 so_21900694_21901318_0 . . . 413
14.1715 so_40377100_40377459_2 . . . 413
14.1716 so_38823513_38823713_0 . . . 413
14.1717 so_22055403_22055550_1 . . . 413
14.1718 so_32025409_32025514_0 . . . 413
14.1719 so_43782361_43782947_0 . . . 413
14.1720 so_33504651_33511131_0 . . . 413
14.1721 so_18534101_18543255_0 . . . 413
14.1722 so_27158330_27158396_0 . . . 413
14.1723 so_6688118_6688599_1 . . . 413
14.1724 so_18517920_18518409_0 . . . 413
14.1725 so_10370713_10370981_0 . . . 413
14.1726 so_24960863_24961075_0 . . . 413
14.1727 so_44174489_44174489_0 . . . 413
14.1728 so_25850629_25852676_0 . . . 413
14.1729 so_27964543_27964716_0 . . . 413
14.1730 so_1819131_1819131_0 . . . 413
14.1731 so_1819131_1825872_0 . . . 413

14.1732 so_2851991_2852234_0 . . . 413
14.1733 so_19964575_19964793_0 . . . 413
14.1734 so_27830761_27883353_0 . . . 413
14.1735 so_34885025_34885535_1 . . . 413
14.1736 so_9549714_9549714_0 . . . 413
14.1737 so_26673081_26673205_1 . . . 413
14.1738 so_28578061_28579464_1 . . . 413
14.1739 so_18899102_18899585_0 . . . 413
14.1740 so_31328754_31328843_0 . . . 413
14.1741 so_24434157_24630425_6 . . . 413
14.1742 so_4894854_4894884_0 . . . 413
14.1743 so_3662450_3662492_0 . . . 413
14.1744 so_38522394_38522440_0 . . . 413
14.1745 so_28667924_28701901_3 . . . 413
14.1746 so_41010924_41011027_0 . . . 413
14.1747 so_26123254_26123293_0 . . . 413
14.1748 so_21598635_21599387_0 . . . 413
14.1749 so_37563745_37564492_0 . . . 413
14.1750 so_27021240_27021316_0 . . . 413
14.1751 so_5728512_5728887_0 . . . 413
14.1752 so_25486033_25486107_0 . . . 413
14.1753 so_26422189_26422452_0 . . . 413
14.1754 so_6735720_6735859_0 . . . 413
14.1755 so_43808848_43809035_0 . . . 413
14.1756 so_26123254_26123293_2 . . . 413
14.1757 so_30415930_30416005_2 . . . 413
14.1758 so_25486033_25486107_2 . . . 413
14.1759 so_25616693_25617043_0 . . . 413
14.1760 so_24106502_24106820_1 . . . 413
14.1761 so_24106502_24106820_3 . . . 413
14.1762 so_4600565_4600746_0 . . . 413
14.1763 so_18174368_18174368_1 . . . 413
14.1764 so_29516206_29516206_0 . . . 413
14.1765 so_3631672_3631724_0 . . . 413
14.1766 so_26673081_26673081_2 . . . 413
14.1767 so_25925164_25925285_0 . . . 413
14.1768 so_3880683_3880683_0 . . . 414
14.1769 so_22258054_22259156_0 . . . 414
14.1770 so_14554288_14562285_0 . . . 414
14.1771 so_12015938_17509696_0 . . . 414
14.1772 so_27481905_27483231_5 . . . 414
14.1773 so_25925551_26005482_6 . . . 414
14.1774 so_25959633_25960570_3 . . . 414
14.1775 so_46473945_46474184_0 . . . 414
14.1776 so_25925164_25925285_1 . . . 414
14.1777 so_24106502_24106820_2 . . . 414
14.1778 so_24106502_24106820_4 . . . 414
14.1779 so_27481905_27483231_7 . . . 414
14.1780 so_13906874_13907192_1 . . . 414
14.1781 so_19345033_19361939_1 . . . 414
14.1782 so_19341360_19341623_1 . . . 414

14.1783 so_18570632_18571345_1 . . . 414
14.1784 so_17132114_17132624_2 . . . 414
14.1785 so_6688118_6688626_1 . . . 414
14.1786 so_25656001_25697507_0 . . . 414
14.1787 so_26673081_26673081_1 . . . 414
14.1788 so_11856637_11859160_1 . . . 414
14.1789 so_37480778_37481069_0 . . . 414
14.1790 so_27021240_27021316_1 . . . 414
14.1791 so_27709461_27709701_0 . . . 414
14.1792 so_25959633_25960570_1 . . . 414
14.1793 so_29616985_38713734_0 . . . 414
14.1794 so_15891106_15891160_1 . . . 415
14.1795 so_1806022_1806038_1 . . . 415
14.1796 so_17118294_17118768_0 . . . 415
14.1797 so_16195273_16195273_0 . . . 415
14.1798 so_30788104_30788225_0 . . . 415
14.1799 so_1662905_1663245_0 . . . 415
14.1800 so_37480778_37481069_2 . . . 415
14.1801 so_25009061_25009130_0 . . . 415
14.1802 so_25397011_25397133_1 . . . 415
14.1803 so_20669394_20803084_0 . . . 415
14.1804 so_38522394_38522440_1 . . . 415
14.1805 so_14554288_14562285_1 . . . 415
14.1806 so_11970707_11970707_0 . . . 415
14.1807 so_3295283_3295425_0 . . . 415
14.1808 so_38823513_38823713_1 . . . 415
14.1809 so_22055403_22055570_0 . . . 415
14.1810 so_32025409_32025514_1 . . . 415
14.1811 so_27158330_27158396_1 . . . 415
14.1812 so_41174161_41174338_0 . . . 415
14.1813 so_29773835_29774235_0 . . . 415
14.1814 so_27777283_27777418_0 . . . 415
14.1815 so_33648155_33648264_1 . . . 415
14.1816 so_18174368_18174368_2 . . . 415
14.1817 so_43982799_43983372_0 . . . 415
14.1818 so_19630138_19630352_1 . . . 415
14.1819 so_183108_183235_0 . . . 415
14.1820 so_27021240_27021316_4 . . . 415
14.1821 so_5728512_5728887_1 . . . 415
14.1822 so_23679525_23679903_1 . . . 415
14.1823 so_19964575_19964793_2 . . . 415
14.1824 so_1622694_1622721_0 . . . 415
14.1825 so_6735720_6735859_1 . . . 415
14.1826 so_28578061_28579464_2 . . . 415
14.1827 so_3483932_3484028_0 . . . 415
14.1828 so_46341681_46935513_0 . . . 415
14.1829 so_11890996_11891953_0 . . . 415
14.1830 so_29230353_29232857_0 . . . 415
14.1831 so_3662450_3662567_0 . . . 415
14.1832 so_37617717_37618181_1 . . . 415
14.1833 so_3295283_3295425_1 . . . 415

14.1834 so_13484308_13484670_1 . . . 415
14.1835 so_24909652_24910057_5 . . . 415
14.1836 so_26422189_26422315_0 . . . 415
14.1837 so_3662450_3662567_1 . . . 415
14.1838 so_25646173_25646388_0 . . . 416
14.1839 so_9491039_9491245_0 . . . 416
14.1840 so_4357854_4357873_2 . . . 416
14.1841 so_36333003_36334378_0 . . . 416
14.1842 so_42932453_42932453_2 . . . 416
14.1843 so_41552366_41554001_0 . . . 416
14.1844 so_17195773_17196371_3 . . . 416
14.1845 so_34506356_34506955_3 . . . 416
14.1846 so_16077430_16077663_0 . . . 416
14.1847 so_15007244_15007521_2 . . . 416
14.1848 so_15342653_15342653_1 . . . 416
14.1849 so_14021243_14021362_3 . . . 416
14.1850 so_34506356_34506955_4 . . . 416
14.1851 so_36558915_36559249_1 . . . 416
14.1852 so_21389403_21389563_0 . . . 416
14.1853 so_24333686_24335198_2 . . . 416
14.1854 so_24333686_24335198_1 . . . 416
14.1855 so_19887425_19887425_0 . . . 416
14.1856 so_24001080_24001172_0 . . . 416
14.1857 so_8569476_8569519_0 . . . 416
14.1858 so_7111957_7112168_2 . . . 416
14.1859 so_8400791_8400791_1 . . . 416
14.1860 so_6304259_6415754_0 . . . 416
14.1861 so_43827514_43827979_0 . . . 416
14.1862 so_43787462_43792468_1 . . . 416
14.1863 so_26983351_26992473_0 . . . 417
14.1864 so_15045879_15077078_0 . . . 417
14.1865 so_11126203_11126203_0 . . . 417
14.1866 so_32054993_32524880_0 . . . 417
14.1867 so_115703_115821_0 . . . 417
14.1868 so_24333686_24335198_8 . . . 417
14.1869 so_24333686_24335198_7 . . . 417
14.1870 so_29576750_29576804_0 . . . 417
14.1871 so_7418553_7418808_0 . . . 417
14.1872 so_17195773_17196371_1 . . . 417
14.1873 so_8752837_8752870_0 . . . 417
14.1874 so_41528905_41528920_1 . . . 417
14.1875 so_3416339_3416740_0 . . . 417
14.1876 so_6304259_6415754_1 . . . 417
14.1877 so_425576_425715_0 . . . 417
14.1878 so_43787462_43792468_2 . . . 417
14.1879 so_3514991_3516147_0 . . . 418
14.1880 so_30788104_30788225_1 . . . 418
14.1881 so_33021636_33021890_0 . . . 418
14.1882 so_33504651_33511131_2 . . . 418
14.1883 so_20816943_20817743_0 . . . 418
14.1884 so_6688118_6688599_0 . . . 418

14.1885 so_33418573_33418573_0 . . . 418
14.1886 so_24326311_24326579_0 . . . 418
14.1887 so_24326311_24326545_0 . . . 418
14.1888 so_22381685_22408491_0 . . . 418
14.1889 so_14151901_14152081_0 . . . 418
14.1890 so_12877931_13204220_0 . . . 418
14.1891 so_12362024_36714283_0 . . . 418
14.1892 so_12362024_36714283_1 . . . 418
14.1893 so_6870885_6870885_0 . . . 418
14.1894 so_6870885_6873644_0 . . . 418
14.1895 so_23411524_23413082_0 . . . 418
14.1896 so_32341278_32346991_0 . . . 418
14.1897 so_12650737_12650941_1 . . . 418
14.1898 so_12650737_12650941_0 . . . 418
14.1899 so_27609522_27609623_0 . . . 418
14.1900 so_38929950_38931484_2 . . . 418
14.1901 so_41915587_41916709_0 . . . 418
14.1902 so_23326403_23326404_0 . . . 418
14.1903 so_23326403_23326404_1 . . . 418
14.1904 so_6870885_6870885_1 . . . 418
14.1905 so_32900588_32900935_0 . . . 418
14.1906 so_16797938_16798012_0 . . . 418
14.1907 so_32895314_32899144_0 . . . 418
14.1908 so_14021243_14021362_0 . . . 418
14.1909 so_38929950_38930793_0 . . . 418
14.1910 so_33453352_33453352_0 . . . 418
14.1911 so_29872879_29879700_0 . . . 418
14.1912 so_9903718_9904241_0 . . . 418
14.1913 so_47142451_47142451_0 . . . 418
14.1914 so_47142451_47142451_1 . . . 418
14.1915 so_23920027_23920027_0 . . . 418
14.1916 so_9473719_9474634_0 . . . 418
14.1917 so_27196600_28525629_0 . . . 418
14.1918 so_9907328_9907600_0 . . . 418
14.1919 so_5013717_14759603_2 . . . 418
14.1920 so_12993219_12993702_0 . . . 418
14.1921 so_39807325_39807325_0 . . . 418
14.1922 so_45524824_45525034_4 . . . 419
14.1923 so_26735919_26736262_0 . . . 419
14.1924 so_13923684_13925256_2 . . . 419
14.1925 so_17580273_17580390_2 . . . 419
14.1926 so_30130834_30131033_0 . . . 419
14.1927 so_11271553_11271636_1 . . . 419
14.1928 so_12335762_12335796_9 . . . 419
14.1929 so_42048621_42084008_0 . . . 419
14.1930 so_13535223_13535223_0 . . . 419
14.1931 so_12340105_12340271_9 . . . 419
14.1932 so_25709653_25709686_0 . . . 419
14.1933 so_28071129_28071356_0 . . . 419
14.1934 so_21389403_21389685_0 . . . 419
14.1935 so_27764647_27764647_0 . . . 419

14.1936 so_30400814_30400814_0 . . . 419
14.1937 so_33099306_33099716_0 . . . 419
14.1938 so_35318293_35318713_1 . . . 419
14.1939 so_7111957_7112168_3 . . . 419
14.1940 so_43735443_43736209_1 . . . 419
14.1941 so_37132632_37132814_1 . . . 419
14.1942 so_30409044_30409055_0 . . . 419
14.1943 so_25311512_25311919_0 . . . 419
14.1944 so_32973787_32974857_1 . . . 419
14.1945 so_25594644_25594741_0 . . . 419
14.1946 so_33056959_33057074_0 . . . 419
14.1947 so_33056959_33057074_1 . . . 419
14.1948 so_26979392_26979615_0 . . . 419
14.1949 so_14078381_14079488_0 . . . 419
14.1950 so_1413158_1413347_0 . . . 419
14.1951 so_27806775_27806775_0 . . . 420
14.1952 so_16473499_16473692_0 . . . 420
14.1953 so_45524824_45524824_0 . . . 420
14.1954 so_24333686_24335198_11 . . . 420
14.1955 so_24333686_24335198_12 . . . 420
14.1956 so_25799846_25799885_0 . . . 420
14.1957 so_46780197_46780630_0 . . . 420
14.1958 so_31343134_31344054_1 . . . 420
14.1959 so_19611478_19611775_0 . . . 420
14.1960 so_21644259_21644259_0 . . . 420
14.1961 so_36850590_36851335_0 . . . 420
14.1962 so_16765179_16765329_2 . . . 420
14.1963 so_20817378_20818023_1 . . . 420
14.1964 so_28065783_28065956_0 . . . 420
14.1965 so_11145015_11145181_0 . . . 421
14.1966 so_1197340_1197490_1 . . . 421
14.1967 so_25661817_25661986_1 . . . 421
14.1968 so_5912689_5912745_2 . . . 421
14.1969 so_31393530_31543319_2 . . . 421
14.1970 so_15125316_15125401_1 . . . 421
14.1971 so_25200678_25201322_0 . . . 421
14.1972 so_25200678_25200887_0 . . . 421
14.1973 so_16910009_16910285_0 . . . 421
14.1974 so_26303241_26303533_0 . . . 421
14.1975 so_15061774_15061929_0 . . . 421
14.1976 so_22989281_22989829_1 . . . 421
14.1977 so_25195429_25195429_1 . . . 421
14.1978 so_25195429_25195429_0 . . . 421
14.1979 so_29635532_29635789_0 . . . 421
14.1980 so_31751138_31751468_0 . . . 421
14.1981 so_24356807_24357265_0 . . . 421
14.1982 so_12860175_12864290_1 . . . 421
14.1983 so_43750053_43750942_0 . . . 421
14.1984 so_24964760_24964971_0 . . . 421
14.1985 so_31343134_31343134_0 . . . 421
14.1986 so_9491973_9492837_3 . . . 421

14.1987 so_15885492_15885633_0 . . . 421
14.1988 so_30530364_30530378_0 . . . 421
14.1989 so_17720424_17720605_1 . . . 421
14.1990 so_24678735_24683940_0 . . . 421
14.1991 so_21936455_21937122_0 . . . 421
14.1992 so_7190424_7190915_1 . . . 421
14.1993 so_10438624_10438746_0 . . . 421
14.1994 so_46984306_46985214_0 . . . 421
14.1995 so_39976307_39976548_0 . . . 421
14.1996 so_46799417_46799684_0 . . . 421
14.1997 so_36997351_36999861_0 . . . 421
14.1998 so_12681622_12681690_0 . . . 421
14.1999 so_16392885_16393281_0 . . . 421
14.2000 so_2797533_2797727_0 . . . 421
14.2001 so_32321006_32321006_2 . . . 421
14.2002 so_21071717_21071880_0 . . . 421
14.2003 so_32973787_32974857_0 . . . 421
14.2004 so_14078381_14079488_1 . . . 421
14.2005 so_18899102_18899102_0 . . . 421
14.2006 so_2342550_2342696_0 . . . 421
14.2007 so_25661817_25661986_0 . . . 422
14.2008 so_15125316_15125401_2 . . . 422
14.2009 so_29635532_29635789_1 . . . 422
14.2010 so_31751138_31751468_1 . . . 422
14.2011 so_3935421_3939222_0 . . . 422
14.2012 so_18557934_18558202_0 . . . 422
14.2013 so_30530364_30530378_1 . . . 422
14.2014 so_24678735_24683940_1 . . . 422
14.2015 so_40826618_40828823_0 . . . 422
14.2016 so_46799417_46799684_1 . . . 422
14.2017 so_36997351_36999861_1 . . . 422
14.2018 so_12681622_12681690_1 . . . 422
14.2019 so_12681622_12681690_2 . . . 422
14.2020 so_2797533_2797727_1 . . . 422
14.2021 so_9582034_9582687_9 . . . 422
14.2022 so_23720937_23721061_0 . . . 422
14.2023 so_27917680_27922878_0 . . . 422
14.2024 so_32788663_32788961_2 . . . 422
14.2025 so_30051718_30051793_0 . . . 422
14.2026 so_13535223_13535253_0 . . . 422
14.2027 so_39239671_39248589_0 . . . 422
14.2028 so_26305368_26305550_2 . . . 423
14.2029 so_25758746_25758850_0 . . . 423
14.2030 so_17667098_17667265_1 . . . 423
14.2031 so_23256676_23257516_0 . . . 423
14.2032 so_16024951_16025045_0 . . . 423
14.2033 so_47428227_47428372_0 . . . 423
14.2034 so_3408469_3408860_0 . . . 423
14.2035 so_32321006_32321006_0 . . . 423
14.2036 so_2342550_2342696_1 . . . 423
14.2037 so_13923684_13925256_3 . . . 423

14.2038 so_28193521_29545974_1 . . . 424
14.2039 so_31128159_31128268_0 . . . 424
14.2040 so_17129543_17129555_1 . . . 424
14.2041 so_27909185_27909355_0 . . . 424
14.2042 so_32909339_32909764_0 . . . 424
14.2043 so_9276389_9276740_0 . . . 424
14.2044 so_31361868_31361934_0 . . . 424
14.2045 so_3236199_3236408_0 . . . 424
14.2046 so_5425065_5425343_0 . . . 424
14.2047 so_40685032_40685032_0 . . . 424
14.2048 so_2537716_2537776_0 . . . 424
14.2049 so_28167880_28168022_0 . . . 424
14.2050 so_37788300_37788300_0 . . . 424
14.2051 so_4881281_4881349_0 . . . 424
14.2052 so_5950457_5950924_0 . . . 424
14.2053 so_3015507_3015554_0 . . . 424
14.2054 so_21644259_21644341_0 . . . 424
14.2055 so_17129543_17129555_0 . . . 424
14.2056 so_4352880_4352936_0 . . . 424
14.2057 so_38374743_38378053_0 . . . 424
14.2058 so_4274828_4274874_0 . . . 424
14.2059 so_46526988_46527608_0 . . . 424
14.2060 so_26045191_26045277_0 . . . 424
14.2061 so_36681857_36682341_0 . . . 424
14.2062 Function template specializations . . . 425
14.2063 so_38814939_38815665_0 . . . 425
14.2064 so_29183791_29188808_6 . . . 425
14.2065 so_5173494_5173876_0 . . . 425
14.2066 so_13521546_13521546_6 . . . 425
14.2067 so_1863751_33734010_0 . . . 425
14.2068 so_41331640_41353091_0 . . . 425
14.2069 so_21573808_36488572_0 . . . 425
14.2070 so_29602638_29611023_0 . . . 425
14.2071 so_36997351_36999343_1 . . . 425
14.2072 so_29056890_29057190_0 . . . 425
14.2073 so_29056890_29057190_1 . . . 425
14.2074 so_24252464_24252692_6 . . . 425
14.2075 so_24252464_24252692_5 . . . 425
14.2076 so_26553803_26557917_1 . . . 425
14.2077 so_13521546_13521546_7 . . . 426
14.2078 so_43304188_44145375_2 . . . 426
14.2079 so_13576845_13576910_0 . . . 426
14.2080 so_44338584_44339464_0 . . . 426
14.2081 so_45628592_45628827_0 . . . 426
14.2082 so_17595988_17603338_0 . . . 426
14.2083 so_13521546_13521546_10 . . . 426
14.2084 so_44419508_44420912_3 . . . 426
14.2085 so_28006408_28140887_0 . . . 426
14.2086 so_41939827_41941916_1 . . . 426
14.2087 so_38158702_42428557_0 . . . 426
14.2088 so_46769112_46769854_1 . . . 426

14.2089 so_27303283_27335365_0 . . . 426
14.2090 so_29417534_29417535_2 . . . 426
14.2091 so_38033124_38033748_2 . . . 426
14.2092 so_40896600_40897181_0 . . . 427
14.2093 so_8513050_8513198_4 . . . 427
14.2094 so_8513050_8513174_3 . . . 427
14.2095 so_6677072_7505108_0 . . . 427
14.2096 so_29330391_29401603_0 . . . 427
14.2097 so_3856712_3856965_0 . . . 427
14.2098 so_30263007_30263103_0 . . . 427
14.2099 so_29987192_29988572_0 . . . 427
14.2100 so_25184932_25185070_0 . . . 427
14.2101 so_23138881_23139850_0 . . . 427
14.2102 so_25101839_25101976_0 . . . 427
14.2103 so_39173519_39181969_1 . . . 427
14.2104 so_14036121_14036121_0 . . . 427
14.2105 so_9838862_9838934_0 . . . 427
14.2106 so_44490086_44490176_0 . . . 427
14.2107 so_7377006_7377096_0 . . . 427
14.2108 so_7377006_7377313_0 . . . 427
14.2109 so_15140315_15140821_1 . . . 427
14.2110 so_12745569_12745920_0 . . . 427
14.2111 so_2953684_39223166_0 . . . 427
14.2112 so_2953684_2953783_0 . . . 427
14.2113 so_25128450_25128825_0 . . . 428
14.2114 so_29417534_29417534_0 . . . 428
14.2115 so_4894854_4894884_1 . . . 428
14.2116 so_16889098_16889154_0 . . . 428
14.2117 so_27633980_27634045_0 . . . 428
14.2118 so_2671460_2671483_0 . . . 428
14.2119 so_11677188_11677315_0 . . . 428
14.2120 so_33417594_33417594_0 . . . 428
14.2121 so_14626033_14628304_1 . . . 428
14.2122 so_30344062_30363491_7 . . . 428
14.2123 so_20268114_20273619_0 . . . 428
14.2124 so_12015938_12016077_0 . . . 428
14.2125 so_38041884_38041884_1 . . . 428
14.2126 so_14098131_14098393_0 . . . 428
14.2127 so_34552056_34716046_3 . . . 428
14.2128 so_44343107_44343107_0 . . . 428
14.2129 so_17116092_17118324_1 . . . 428
14.2130 so_17116092_17118324_3 . . . 428
14.2131 so_17116092_17118324_0 . . . 428
14.2132 so_17116092_17118324_2 . . . 428
14.2133 so_15501470_15502379_0 . . . 429
14.2134 so_24617413_24617413_4 . . . 429
14.2135 so_24617413_24617413_1 . . . 429
14.2136 so_30227277_30230036_1 . . . 429
14.2137 so_17116092_17118324_4 . . . 429
14.2138 so_30448380_35169219_0 . . . 429
14.2139 so_24617413_24617413_8 . . . 429

14.2140 so_24617413_24617413_9 . . . 429
14.2141 so_34876969_34877144_2 . . . 429
14.2142 so_31150922_31150948_0 . . . 429
14.2143 so_22281205_22281608_4 . . . 429
14.2144 so_19083709_19083981_1 . . . 429
14.2145 so_15501470_15502379_2 . . . 429
14.2146 so_21379666_21380433_0 . . . 429
14.2147 so_13638623_13638743_0 . . . 429
14.2148 so_17116092_17118324_5 . . . 429
14.2149 so_30344062_30363491_6 . . . 430
14.2150 so_26443573_26443573_1 . . . 430
14.2151 so_26443573_26443573_0 . . . 430
14.2152 so_8549406_8549824_0 . . . 430
14.2153 so_12654067_12654277_1 . . . 430
14.2154 so_17116092_17118324_6 . . . 430
14.2155 so_45584616_45585426_1 . . . 430
14.2156 so_38966207_38966207_0 . . . 430
14.2157 so_38966207_38966413_0 . . . 430
14.2158 so_33665208_33667546_0 . . . 430
14.2159 so_14846511_14849163_2 . . . 430
14.2160 so_14648434_14649812_0 . . . 430
14.2161 so_15462336_15462337_1 . . . 430
14.2162 so_22368022_22368022_5 . . . 430
14.2163 so_22368022_22368022_4 . . . 430
14.2164 so_22368022_22368022_2 . . . 430
14.2165 so_22368022_22368022_1 . . . 430
14.2166 so_8549406_8549824_1 . . . 430
14.2167 so_12654067_12654277_2 . . . 430
14.2168 so_10377183_10377831_0 . . . 430
14.2169 so_27301135_27301428_4 . . . 430
14.2170 so_24503031_24503406_0 . . . 430
14.2171 so_24503031_24503406_1 . . . 430
14.2172 so_17142130_17147477_9 . . . 430
14.2173 so_17142130_17147477_12 . . . 430
14.2174 so_17142130_17147477_11 . . . 430
14.2175 so_17142130_17147477_7 . . . 430
14.2176 so_17142130_17147477_13 . . . 430
14.2177 so_17142130_17147477_5 . . . 430
14.2178 so_17142130_17147477_8 . . . 430
14.2179 so_17142130_17147477_6 . . . 430
14.2180 so_17142130_17147477_10 . . . 430
14.2181 so_17071686_17071736_2 . . . 430
14.2182 so_17071686_17071736_0 . . . 430
14.2183 so_4131091_4131097_0 . . . 430
14.2184 so_16206626_16210319_0 . . . 430
14.2185 so_21602957_21605768_6 . . . 430
14.2186 so_11856637_11859160_3 . . . 430
14.2187 so_42985624_42986059_0 . . . 430
14.2188 so_32017004_32017587_1 . . . 430
14.2189 so_31673850_31673850_0 . . . 430
14.2190 so_24053998_24054690_4 . . . 430

14.2191 so_24053998_24054690_5 . . . 430
14.2192 so_24053998_24054690_3 . . . 430
14.2193 so_24053998_24054690_2 . . . 430
14.2194 so_30969209_30969314_0 . . . 430
14.2195 so_26672776_26672776_3 . . . 430
14.2196 so_8984013_8984379_1 . . . 430
14.2197 so_8984013_8984268_3 . . . 430
14.2198 so_8984013_8984268_7 . . . 430
14.2199 so_8984013_8984268_9 . . . 430
14.2200 so_8984013_8984268_8 . . . 430
14.2201 so_8984013_8984379_0 . . . 430
14.2202 so_27359746_27360064_0 . . . 430
14.2203 so_14626033_14628304_0 . . . 430
14.2204 so_23741529_23743884_0 . . . 430
14.2205 so_8955052_8955220_1 . . . 430
14.2206 so_34488559_34488782_1 . . . 430
14.2207 so_30676839_32572451_3 . . . 430
14.2208 so_30676839_32572451_2 . . . 430
14.2209 so_30676839_32572451_1 . . . 430
14.2210 so_13809359_13810428_0 . . . 430
14.2211 so_540748_540850_2 . . . 430
14.2212 so_12015938_12016077_1 . . . 430
14.2213 so_38966207_38966413_1 . . . 430
14.2214 so_6321191_6321191_0 . . . 430
14.2215 so_17853767_17853925_0 . . . 430
14.2216 so_14846511_14849163_3 . . . 430
14.2217 so_14808663_14836911_1 . . . 430
14.2218 so_14808663_14836911_0 . . . 430
14.2219 so_14648434_14649812_2 . . . 430
14.2220 so_14648434_14649812_1 . . . 430
14.2221 so_35289985_35291820_0 . . . 430
14.2222 so_14098131_14098393_1 . . . 430
14.2223 so_27943503_27943618_0 . . . 430
14.2224 so_22368022_22368023_4 . . . 430
14.2225 so_22368022_22368023_5 . . . 430
14.2226 so_22368022_22368023_6 . . . 430
14.2227 so_22368022_22368023_3 . . . 430
14.2228 so_22368022_22368023_8 . . . 430
14.2229 so_22368022_22368023_1 . . . 430
14.2230 so_22368022_22368023_2 . . . 430
14.2231 so_22368022_22368023_7 . . . 430
14.2232 so_15260685_15260685_1 . . . 430
14.2233 so_15260685_15260685_0 . . . 430
14.2234 so_30855339_30855588_0 . . . 430
14.2235 so_24909652_24910057_3 . . . 430
14.2236 so_17702250_17703972_0 . . . 430
14.2237 so_37701321_37701321_0 . . . 430
14.2238 so_26949290_26949793_0 . . . 430
14.2239 so_29807455_29807455_0 . . . 430
14.2240 so_35879510_35879510_2 . . . 430
14.2241 so_35879510_35879765_1 . . . 430

14.2242 so_15460325_15462180_0 . . . 430
14.2243 so_27678981_27679361_0 . . . 430
14.2244 so_21352507_21352605_1 . . . 430
14.2245 so_8549406_8549824_2 . . . 430
14.2246 so_39758335_39758335_0 . . . 430
14.2247 so_12654067_12654277_3 . . . 430
14.2248 so_3951209_3952874_10 . . . 430
14.2249 so_46596400_46596400_0 . . . 430
14.2250 so_14866360_14866360_0 . . . 430
14.2251 so_35677263_35677739_0 . . . 430
14.2252 so_17359235_17360707_1 . . . 430
14.2253 so_2733377_2770326_0 . . . 430
14.2254 so_19516159_19516701_0 . . . 430
14.2255 so_8026579_8026579_0 . . . 430
14.2256 so_39980445_39982793_0 . . . 430
14.2257 so_39980445_39982793_1 . . . 430
14.2258 so_43789366_43789366_0 . . . 430
14.2259 so_13257103_13257161_0 . . . 430
14.2260 so_13257103_13257161_1 . . . 430
14.2261 so_37425307_37425433_0 . . . 432
14.2262 so_23550135_23553430_0 . . . 432
14.2263 so_41401092_41401279_0 . . . 432
14.2264 so_24059738_24060303_0 . . . 432
14.2265 so_18851918_18852262_0 . . . 432
14.2266 so_11073423_11073669_0 . . . 432
14.2267 so_36640883_36640883_0 . . . 432
14.2268 so_8513050_8513198_0 . . . 432
14.2269 so_23186775_23186975_0 . . . 432
14.2270 so_28370970_28371763_0 . . . 432
14.2271 so_20059061_20059253_0 . . . 432
14.2272 so_17496268_17496410_0 . . . 432
14.2273 so_30339922_30340999_0 . . . 432
14.2274 so_30344062_30363491_3 . . . 432
14.2275 so_32089214_32089365_2 . . . 432
14.2276 so_35361408_35361496_0 . . . 432
14.2277 so_22636151_22638424_1 . . . 432
14.2278 so_22636151_22638424_0 . . . 432
14.2279 so_33948090_33948222_0 . . . 432
14.2280 so_5173494_5173876_1 . . . 432
14.2281 so_1863751_33734010_1 . . . 432
14.2282 so_37955601_37955601_1 . . . 432
14.2283 so_31258811_31258865_2 . . . 432
14.2284 so_39239671_39248589_1 . . . 432
14.2285 so_7857985_7858222_0 . . . 432
14.2286 so_37645347_37645712_0 . . . 432
14.2287 so_21573808_36488572_2 . . . 432
14.2288 so_21573808_36488572_1 . . . 432
14.2289 so_22300620_22415912_0 . . . 432
14.2290 so_28006408_28140887_1 . . . 432
14.2291 so_29867841_29868348_1 . . . 432
14.2292 so_46997165_46999091_1 . . . 432

14.2293 so_29602638_29611023_1 . . . 432
14.2294 so_27303283_27335365_1 . . . 432
14.2295 so_38629601_38635008_0 . . . 432
14.2296 so_27976403_28391532_0 . . . 432
14.2297 so_40704605_40705233_2 . . . 432
14.2298 so_16910595_16911547_0 . . . 432
14.2299 so_31292403_31293774_0 . . . 432
14.2300 so_32515183_32515264_1 . . . 432
14.2301 so_11931452_11931607_3 . . . 433
14.2302 so_11931452_11931607_4 . . . 433
14.2303 so_21070032_21070985_5 . . . 433
14.2304 so_21070032_21070985_2 . . . 433
14.2305 so_21070032_21070985_3 . . . 433
14.2306 so_21070032_21070985_4 . . . 433
14.2307 so_39150553_39150553_2 . . . 433
14.2308 so_32089214_32089365_4 . . . 433
14.2309 so_5173494_5173876_3 . . . 433
14.2310 so_5173494_5173876_4 . . . 433
14.2311 so_5173494_5173876_5 . . . 433
14.2312 so_31872991_31873347_3 . . . 433
14.2313 so_6443230_6444871_1 . . . 433
14.2314 so_38663166_38664084_2 . . . 433
14.2315 so_38663166_38664084_3 . . . 433
14.2316 so_38663166_38664084_1 . . . 433
14.2317 so_7797839_7797957_0 . . . 433
14.2318 so_16794633_16794743_3 . . . 433
14.2319 so_16794633_16794743_1 . . . 433
14.2320 so_16794633_16794743_2 . . . 433
14.2321 so_32130745_32130883_1 . . . 433
14.2322 so_24634416_24634807_0 . . . 433
14.2323 so_38814939_38815665_1 . . . 433
14.2324 so_16192796_16192955_0 . . . 433
14.2325 so_11931452_11931607_5 . . . 433
14.2326 so_32282705_32284702_1 . . . 433
14.2327 so_21070032_21070985_6 . . . 433
14.2328 so_21070032_21070985_7 . . . 433
14.2329 so_33904462_33904463_1 . . . 433
14.2330 so_30339922_30340999_1 . . . 433
14.2331 so_15501470_15504781_0 . . . 433
14.2332 so_29756518_29756644_0 . . . 433
14.2333 so_20364297_20378876_3 . . . 433
14.2334 so_20364297_20378876_2 . . . 433
14.2335 so_11279647_11281056_1 . . . 433
14.2336 so_29183791_29188808_7 . . . 433
14.2337 so_19759136_19759603_0 . . . 433
14.2338 so_36843699_36849967_0 . . . 433
14.2339 so_5173494_5173876_6 . . . 433
14.2340 so_45645244_45645456_0 . . . 433
14.2341 so_39855810_39855810_0 . . . 433
14.2342 so_39855810_39855810_1 . . . 433
14.2343 so_41331640_41353091_1 . . . 433

14.2344 so_33426038_33426038_1 . . . 433
14.2345 so_28028606_28028747_1 . . . 433
14.2346 so_38663166_38664084_4 . . . 433
14.2347 so_40819058_40819119_0 . . . 433
14.2348 so_35666265_35666626_0 . . . 433
14.2349 so_39552272_39552345_1 . . . 433
14.2350 so_31804946_31810042_2 . . . 433
14.2351 so_27047795_27048079_2 . . . 433
14.2352 so_16910595_16911547_1 . . . 433
14.2353 so_31292403_31293774_1 . . . 433
14.2354 so_29851098_29851257_0 . . . 433
14.2355 so_8513050_8513198_1 . . . 433
14.2356 so_8513050_8513174_2 . . . 433
14.2357 so_8513050_8513198_2 . . . 433
14.2358 so_8513050_8513198_3 . . . 433
14.2359 so_27400339_27401103_0 . . . 433
14.2360 so_29245209_29245344_1 . . . 433
14.2361 so_29245209_29245344_3 . . . 433
14.2362 so_29245209_29245344_0 . . . 433
14.2363 so_29245209_29245344_2 . . . 433
14.2364 so_45751978_45753280_0 . . . 433
14.2365 so_22848619_22848951_2 . . . 433
14.2366 so_22848619_22848951_3 . . . 433
14.2367 so_22848619_22848951_0 . . . 433
14.2368 so_15501470_15504781_1 . . . 433
14.2369 so_11886930_11887041_1 . . . 433
14.2370 so_18989869_18991898_0 . . . 433
14.2371 so_18989869_18991898_3 . . . 433
14.2372 so_18989869_18991898_1 . . . 433
14.2373 so_18989869_18991898_2 . . . 433
14.2374 so_14971536_14972228_0 . . . 433
14.2375 so_14971536_14972228_2 . . . 433
14.2376 so_3859517_3859819_0 . . . 433
14.2377 so_25184932_25185070_1 . . . 433
14.2378 so_2392515_2407544_0 . . . 433
14.2379 so_39173519_39181969_0 . . . 433
14.2380 so_31292403_31293774_2 . . . 433
14.2381 so_23550135_23553430_1 . . . 434
14.2382 so_19612378_19612457_0 . . . 434
14.2383 so_29245209_29245344_4 . . . 434
14.2384 so_4947618_4947668_1 . . . 434
14.2385 so_14971536_14972228_3 . . . 434
14.2386 so_30240851_30241368_1 . . . 434
14.2387 so_2140025_2140182_0 . . . 434
14.2388 so_2392515_2407544_1 . . . 434
14.2389 so_2392515_2412209_0 . . . 434
14.2390 so_25745759_25758367_8 . . . 434
14.2391 so_11961905_11962016_0 . . . 434
14.2392 so_40343663_40350274_2 . . . 434
14.2393 so_40343663_40350274_1 . . . 434
14.2394 so_40343663_40350274_0 . . . 434

14.2395 so_13527161_13527268_1 . . . 434
14.2396 so_33624891_39406298_2 . . . 434
14.2397 so_33624891_39406298_0 . . . 434
14.2398 so_33624891_39406298_1 . . . 434
14.2399 so_44419508_44420912_2 . . . 434
14.2400 so_44419508_44420912_0 . . . 434
14.2401 so_30623939_30624239_1 . . . 434
14.2402 so_30623939_30624239_2 . . . 434
14.2403 so_30623939_30624239_0 . . . 434
14.2404 so_29417534_29417535_0 . . . 434
14.2405 so_40356990_40357386_1 . . . 434
14.2406 so_40356990_40357386_0 . . . 434
14.2407 so_31316229_31316428_0 . . . 434
14.2408 so_14945564_14945668_0 . . . 435
14.2409 so_30277891_30277891_0 . . . 435
14.2410 so_24390911_24392099_1 . . . 435
14.2411 so_24390911_24392067_0 . . . 435
14.2412 so_39319745_39319745_1 . . . 435
14.2413 so_41175957_41176278_0 . . . 435
14.2414 so_39150553_39150553_0 . . . 435
14.2415 so_39659011_39659230_2 . . . 435
14.2416 so_31569716_31570089_0 . . . 435
14.2417 so_39150553_39150553_1 . . . 435
14.2418 so_25745759_25758367_7 . . . 435
14.2419 so_17005985_17008568_0 . . . 436
14.2420 so_17005985_17005985_0 . . . 436
14.2421 so_17005985_17008568_3 . . . 436
14.2422 so_17005985_17008568_2 . . . 436
14.2423 so_31051189_31051322_0 . . . 436
14.2424 so_42957146_42957361_2 . . . 436
14.2425 so_17412686_17412835_0 . . . 436
14.2426 so_28308599_28309212_0 . . . 436
14.2427 so_20041044_20057950_0 . . . 436
14.2428 so_19365571_19406528_2 . . . 436
14.2429 so_43304188_44145375_4 . . . 436
14.2430 so_47495384_47495429_0 . . . 436
14.2431 so_47495384_47495429_1 . . . 436
14.2432 so_31051189_31051322_2 . . . 436
14.2433 so_42957146_42957361_3 . . . 436
14.2434 so_31497357_31735144_2 . . . 436
14.2435 so_32373979_32373979_1 . . . 436
14.2436 so_32373979_32373979_2 . . . 436
14.2437 so_32373979_32373979_4 . . . 436
14.2438 so_32373979_32373979_3 . . . 436
14.2439 so_31394260_31735126_0 . . . 436
14.2440 so_31051189_31051322_3 . . . 436
14.2441 so_32373979_32373979_5 . . . 436
14.2442 so_39355629_39356164_1 . . . 436
14.2443 so_31563235_31886631_1 . . . 436
14.2444 so_32368890_32371173_1 . . . 436
14.2445 so_32368890_32371173_2 . . . 436

14.2446 so_32368890_32371173_3 . . . 436
14.2447 so_11703553_11703570_1 . . . 436
14.2448 so_11703553_11703570_0 . . . 436
14.2449 so_11703553_11703570_2 . . . 436
14.2450 so_14453829_14454256_1 . . . 436
14.2451 so_28519704_28519731_3 . . . 436
14.2452 so_28519704_28519731_2 . . . 436
14.2453 so_26317798_26318330_3 . . . 436
14.2454 so_31051189_31051322_4 . . . 437
14.2455 so_36242567_36242724_0 . . . 437
14.2456 so_36293088_36293522_1 . . . 437
14.2457 so_14180170_25789864_0 . . . 437
14.2458 so_41902914_41903588_1 . . . 437
14.2459 so_41902914_41903588_0 . . . 437
14.2460 so_31051189_31051322_5 . . . 437
14.2461 so_33426038_33436621_0 . . . 437
14.2462 so_32373979_32373979_6 . . . 437
14.2463 so_32373979_32375678_3 . . . 437
14.2464 so_46309228_46311003_2 . . . 437
14.2465 so_32368890_32371173_4 . . . 437
14.2466 so_32368890_32371173_5 . . . 437
14.2467 so_31394260_31735126_3 . . . 437
14.2468 so_44346513_44346749_0 . . . 437
14.2469 so_9882871_9882871_0 . . . 437
14.2470 so_22248157_22248922_0 . . . 437
14.2471 so_2577623_2577764_4 . . . 437
14.2472 so_31051189_31051322_11 . . . 437
14.2473 so_31051189_31051322_6 . . . 437
14.2474 so_31051189_31051322_8 . . . 437
14.2475 so_31051189_31051322_13 . . . 437
14.2476 so_16999039_16999060_1 . . . 437
14.2477 so_16999039_16999060_0 . . . 437
14.2478 so_15702143_15710873_1 . . . 437
14.2479 so_15702143_15706659_2 . . . 437
14.2480 so_15702143_15702143_0 . . . 437
14.2481 so_15702143_15706659_3 . . . 437
14.2482 so_15702143_15706659_5 . . . 437
14.2483 so_15702143_15710873_0 . . . 437
14.2484 so_15702143_15706659_6 . . . 437
14.2485 so_2577623_2577764_5 . . . 437
14.2486 so_14453829_14454256_5 . . . 437
14.2487 so_39286114_39295906_2 . . . 438
14.2488 so_11499863_11501098_0 . . . 438
14.2489 so_17412686_17412835_1 . . . 438
14.2490 so_17699858_17708667_4 . . . 438
14.2491 so_8404210_8576751_0 . . . 438
14.2492 so_33426038_33426038_0 . . . 438
14.2493 so_17736337_17807344_0 . . . 438
14.2494 so_24390911_24392099_3 . . . 438
14.2495 so_41027701_41075328_1 . . . 438
14.2496 so_34552056_34552056_2 . . . 438

14.2497 so_41143524_41184810_0 . . . 438
14.2498 so_31497357_31735144_4 . . . 438
14.2499 so_34714752_34714752_3 . . . 438
14.2500 so_9037940_9040338_0 . . . 438
14.2501 so_26709862_26710071_2 . . . 438
14.2502 so_515788_516742_0 . . . 438
14.2503 so_24390911_24392099_4 . . . 438
14.2504 so_23550135_23553430_2 . . . 439
14.2505 so_34876969_34877144_4 . . . 439
14.2506 so_35875829_35877464_1 . . . 439
14.2507 so_26842738_26842877_2 . . . 439
14.2508 so_42240849_42241026_1 . . . 439
14.2509 so_42240849_42241015_1 . . . 439
14.2510 so_6593821_6594440_2 . . . 439
14.2511 so_31429127_31438775_1 . . . 439
14.2512 so_17388317_17388446_0 . . . 439
14.2513 so_31066712_31067392_0 . . . 439
14.2514 so_31392020_31392265_1 . . . 439
14.2515 so_41027701_41075328_3 . . . 439
14.2516 so_13527161_13527268_0 . . . 439
14.2517 so_12668766_12669170_0 . . . 439
14.2518 so_26089189_26089329_0 . . . 439
14.2519 so_30765151_30765293_0 . . . 439
14.2520 so_41938104_41938247_0 . . . 439
14.2521 so_17699858_17708667_0 . . . 439
14.2522 so_40356990_40357386_2 . . . 439
14.2523 so_17582667_17582667_0 . . . 439
14.2524 so_18498957_18498957_0 . . . 439
14.2525 so_34876969_34877144_6 . . . 439
14.2526 so_34876969_34877144_7 . . . 439
14.2527 so_46054228_46054902_1 . . . 439
14.2528 so_32972132_32973315_0 . . . 439
14.2529 so_32188099_32188647_7 . . . 439
14.2530 so_32188099_32188647_3 . . . 439
14.2531 so_32188099_32188647_5 . . . 439
14.2532 so_32188099_32188647_6 . . . 439
14.2533 so_32188099_32188647_1 . . . 439
14.2534 so_32188099_32188647_2 . . . 439
14.2535 so_32188099_32188647_4 . . . 439
14.2536 so_12431495_12431603_0 . . . 439
14.2537 so_12431495_12431810_0 . . . 439
14.2538 so_35875829_35877464_2 . . . 439
14.2539 so_37493026_38714121_1 . . . 439
14.2540 so_37493026_38714121_0 . . . 439
14.2541 so_26842738_26842877_1 . . . 439
14.2542 so_31892608_31892671_1 . . . 439
14.2543 so_47495384_47495416_0 . . . 439
14.2544 so_39104749_39110987_0 . . . 439
14.2545 so_42240849_42241015_2 . . . 439
14.2546 so_42240849_42241026_2 . . . 439
14.2547 so_42244250_42246348_0 . . . 439

14.2548 so_29941499_29941820_0 . . . 439
14.2549 so_45769950_45800815_2 . . . 439
14.2550 so_6593821_6594440_4 . . . 439
14.2551 so_16602175_16602198_2 . . . 439
14.2552 so_19128646_19128780_1 . . . 439
14.2553 so_15443675_15443758_1 . . . 439
14.2554 so_30344062_30363491_2 . . . 439
14.2555 so_17332286_17335098_0 . . . 439
14.2556 so_16093073_16093692_1 . . . 439
14.2557 so_31429127_31438775_2 . . . 439
14.2558 so_31066712_31067392_2 . . . 439
14.2559 so_31392020_31392265_2 . . . 439
14.2560 so_32661088_32661153_0 . . . 439
14.2561 so_25179414_25184733_0 . . . 439
14.2562 so_30649469_30650191_1 . . . 439
14.2563 so_25738069_25738124_0 . . . 439
14.2564 so_28290792_28290885_2 . . . 439
14.2565 so_3830491_3830564_0 . . . 439
14.2566 so_24277974_24278483_0 . . . 439
14.2567 so_45816776_45816776_0 . . . 439
14.2568 so_31258811_31258865_0 . . . 439
14.2569 so_32204901_32205820_2 . . . 439
14.2570 so_32204901_32205820_4 . . . 439
14.2571 so_32204901_32205820_5 . . . 439
14.2572 so_37846565_37846954_0 . . . 439
14.2573 so_12668766_12669170_2 . . . 439
14.2574 so_30240851_30241159_8 . . . 439
14.2575 so_30240851_30241159_9 . . . 439
14.2576 so_30240851_30241159_6 . . . 439
14.2577 so_30240851_30241159_5 . . . 439
14.2578 so_30240851_30241159_4 . . . 439
14.2579 so_30240851_30241159_2 . . . 439
14.2580 so_30240131_30241944_4 . . . 439
14.2581 so_37645347_37645712_1 . . . 439
14.2582 so_29593207_29593207_1 . . . 439
14.2583 so_29593207_29593207_0 . . . 439
14.2584 so_20041044_20057950_3 . . . 439
14.2585 so_30765151_30765293_1 . . . 439
14.2586 so_17699858_17708667_2 . . . 439
14.2587 so_41939827_41941916_0 . . . 439
14.2588 so_2140025_2140182_2 . . . 439
14.2589 so_15031198_15032236_1 . . . 439
14.2590 so_15031198_15032236_4 . . . 439
14.2591 so_15031198_15032236_2 . . . 439
14.2592 so_15031198_15032236_8 . . . 439
14.2593 so_15031198_15032236_3 . . . 439
14.2594 so_15031198_15032236_6 . . . 439
14.2595 so_15031198_15032236_7 . . . 439
14.2596 so_15031198_15032236_10 . . . 439
14.2597 so_15031198_15032236_9 . . . 439
14.2598 so_15031198_15032236_5 . . . 439

14.2599 so_29762813_29762987_0 . . . 439
14.2600 so_17807124_17936554_0 . . . 439
14.2601 so_29417534_29417535_1 . . . 439
14.2602 so_38629601_38635008_2 . . . 439
14.2603 so_27976403_27976798_2 . . . 439
14.2604 so_19365571_19406528_5 . . . 439
14.2605 so_37969651_37970907_1 . . . 439
14.2606 so_6593821_6594440_6 . . . 439
14.2607 so_23362391_23430522_1 . . . 439
14.2608 so_1268504_1268915_1 . . . 439
14.2609 so_27976403_27976798_3 . . . 439
14.2610 so_17736337_17807344_2 . . . 440
14.2611 so_17736337_17807344_1 . . . 440
14.2612 so_29183791_29188808_1 . . . 440
14.2613 so_13521163_13738472_0 . . . 440
14.2614 so_34889583_34889583_0 . . . 440
14.2615 so_17807124_17808857_0 . . . 440
14.2616 so_11073423_11073815_0 . . . 441
14.2617 so_25126987_31916090_0 . . . 441
14.2618 so_20874388_20874570_0 . . . 441
14.2619 so_6757691_6757830_2 . . . 441
14.2620 so_35486160_35486160_1 . . . 441
14.2621 so_27230131_27230507_0 . . . 441
14.2622 so_37955601_37955601_2 . . . 441
14.2623 so_33426038_33426038_2 . . . 441
14.2624 so_13521163_13738472_1 . . . 441
14.2625 so_35403116_35403116_0 . . . 441
14.2626 so_17277487_17277945_0 . . . 441
14.2627 so_37924796_37926042_1 . . . 441
14.2628 so_40704605_40705233_3 . . . 441
14.2629 so_41046986_41049584_0 . . . 441
14.2630 so_44340024_44345747_1 . . . 441
14.2631 so_45460916_45461013_0 . . . 441
14.2632 so_42558976_42559381_0 . . . 441
14.2633 so_45790970_45791389_0 . . . 441
14.2634 so_33904462_33904463_3 . . . 441
14.2635 so_33904462_33904463_4 . . . 441
14.2636 so_29183791_29188808_5 . . . 441
14.2637 so_22560325_22566565_1 . . . 441
14.2638 so_22560325_22566565_0 . . . 441
14.2639 so_22560325_22566565_2 . . . 441
14.2640 so_35139955_35160075_5 . . . 441
14.2641 so_35139955_35160075_7 . . . 441
14.2642 so_1447141_1447360_0 . . . 442
14.2643 so_13910747_13912666_0 . . . 442
14.2644 so_44263503_44265134_0 . . . 442
14.2645 so_44415152_44505505_0 . . . 442
14.2646 so_33010352_33011069_1 . . . 443
14.2647 so_46555186_46555969_0 . . . 443
14.2648 so_37644128_37644422_0 . . . 443
14.2649 so_24826139_24826604_1 . . . 443

14.2650 so_35139955_35160075_6 . . . 443
14.2651 so_35139955_35160075_1 . . . 443
14.2652 so_47495384_47495416_1 . . . 444
14.2653 so_7452849_7465863_0 . . . 444
14.2654 so_24120510_24120620_0 . . . 444
14.2655 so_9628974_9629009_0 . . . 444
14.2656 so_26689307_26689568_1 . . . 445
14.2657 so_21602957_21605768_5 . . . 445
14.2658 so_19459283_19571951_0 . . . 445
14.2659 so_18283851_18283933_0 . . . 445
14.2660 so_13521546_13537141_1 . . . 445
14.2661 so_26689307_26689568_2 . . . 445
14.2662 so_13521546_13537141_2 . . . 445
14.2663 so_18899383_18899438_0 . . . 445
14.2664 so_8513050_8513174_1 . . . 445
14.2665 so_39286114_39287634_2 . . . 445
14.2666 so_39286114_39287634_0 . . . 445
14.2667 so_24036357_24043642_0 . . . 445
14.2668 so_32788819_32788819_1 . . . 445
14.2669 so_41411009_41412882_5 . . . 445
14.2670 so_3723630_3723905_1 . . . 445
14.2671 so_20659000_20659598_2 . . . 445
14.2672 so_24909652_24910057_1 . . . 445
14.2673 so_28308599_28308748_0 . . . 445
14.2674 so_15460325_15460325_0 . . . 445
14.2675 so_15208198_15208198_0 . . . 445
14.2676 so_46526988_46527707_0 . . . 445
14.2677 so_28184888_28184888_0 . . . 445
14.2678 so_24036357_24043642_2 . . . 446
14.2679 so_37480778_37481069_1 . . . 446

15 Exception handling **448**
15.1 so_16794972_16795212_0 . . . 448
15.2 so_11458830_11458958_1 . . . 448
15.3 so_11458830_11458958_0 . . . 448
15.4 so_27959696_27959839_0 . . . 448
15.5 so_6756931_6757017_0 . . . 449
15.6 so_32866810_32867482_0 . . . 449
15.7 Throwing an exception . . . 449
15.8 so_17633252_17633556_1 . . . 449
15.9 so_41811023_41811023_1 . . . 449
15.10 so_6185957_6186019_1 . . . 450
15.11 so_12887495_12887534_0 . . . 450
15.12 so_34727228_34727585_0 . . . 450
15.13 so_3561571_3561752_0 . . . 450
15.14 so_16677784_16678377_0 . . . 450
15.15 so_16432959_16433015_0 . . . 450
15.16 so_21720579_21722226_1 . . . 450
15.17 so_6352438_6352438_0 . . . 450
15.18 so_33241984_33242422_1 . . . 450
15.19 so_17785918_17786100_0 . . . 450

15.20 so_25534628_25534823_0 . . . 450
15.21 so_21644735_21645818_0 . . . 450
15.22 so_32761132_32761132_1 . . . 450
15.23 so_32761132_32761132_0 . . . 450
15.24 so_26806857_26806985_2 . . . 450
15.25 so_32759433_32761541_0 . . . 450
15.26 so_33387545_33388006_0 . . . 450
15.27 so_16938951_16939177_0 . . . 450
15.28 so_31526865_31527020_1 . . . 450
15.29 so_31526865_31526975_0 . . . 450
15.30 so_31526865_31527020_0 . . . 450
15.31 so_28875810_28876101_0 . . . 450
15.32 so_41811023_41811023_2 . . . 450
15.33 so_41811023_41811023_0 . . . 450
15.34 so_1855773_27172624_0 . . . 450
15.35 so_12690698_12691011_1 . . . 450
15.36 so_7262818_7263017_2 . . . 450
15.37 so_7262818_7263318_0 . . . 450
15.38 so_7262818_7263017_1 . . . 450
15.39 so_43799790_43799790_2 . . . 450
15.40 so_43799790_43802017_0 . . . 450
15.41 so_16677784_16678377_1 . . . 450
15.42 so_21720579_21722226_0 . . . 450
15.43 so_21644735_21645818_1 . . . 450
15.44 so_20966681_20966744_0 . . . 450
15.45 so_27259652_27259902_0 . . . 450
15.46 so_33387545_33387655_0 . . . 450
15.47 so_33387545_33387659_1 . . . 450
15.48 so_45497684_45497684_0 . . . 450
15.49 so_28353708_28353749_1 . . . 450
15.50 so_8597039_8597527_0 . . . 450
15.51 so_16432959_16433015_1 . . . 450
15.52 so_33241984_33242688_2 . . . 450
15.53 so_17785918_17786100_1 . . . 450
15.54 so_25534628_25534823_1 . . . 450
15.55 so_12824176_12826028_2 . . . 450
15.56 so_17485827_17487841_0 . . . 450
15.57 so_33387545_33387659_2 . . . 450
15.58 so_5840692_5840934_1 . . . 450
15.59 so_22471925_22472200_0 . . . 450
15.60 so_10103645_10103838_0 . . . 450
15.61 so_45035913_45036404_1 . . . 450
15.62 so_10436959_10437245_0 . . . 450
15.63 so_6864011_6864720_2 . . . 450
15.64 so_27743476_27748693_3 . . . 450
15.65 so_32945739_32946005_0 . . . 451
15.66 so_32945739_32946005_1 . . . 451
15.67 so_25638907_25640101_2 . . . 451
15.68 so_26883615_26883786_0 . . . 451
15.69 so_26883615_26883787_1 . . . 451
15.70 so_18467789_18467835_0 . . . 451

15.71 so_18467789_18467835_1 . . . 451
15.72 so_1481612_1481645_0 . . . 451
15.73 so_14775411_14776362_0 . . . 451
15.74 so_23360505_23361149_1 . . . 451
15.75 so_8597039_8597527_3 . . . 451
15.76 so_11308864_11309140_0 . . . 451
15.77 so_44773631_44773683_1 . . . 451
15.78 so_32945739_32946005_3 . . . 451
15.79 so_5378677_5378719_0 . . . 451
15.80 so_1310790_1310911_0 . . . 451
15.81 so_15449629_15449689_0 . . . 451
15.82 so_5766006_5766015_0 . . . 451
15.83 so_2288692_2288727_0 . . . 451
15.84 so_651154_651188_0 . . . 451
15.85 so_651154_651206_2 . . . 451
15.86 so_46190775_46190814_0 . . . 451
15.87 so_8481783_8500265_0 . . . 451
15.88 so_11308864_11309140_1 . . . 451
15.89 Constructors and destructors . . . 451
15.90 so_33689053_33689626_0 . . . 451
15.91 so_4042629_4042907_0 . . . 451
15.92 so_28588671_28588809_0 . . . 451
15.93 so_45220063_45220505_0 . . . 451
15.94 so_34673460_34673460_0 . . . 451
15.95 so_27337324_27337344_0 . . . 451
15.96 so_16870789_16870814_1 . . . 451
15.97 so_8311457_8311472_1 . . . 451
15.98 so_22135946_22136015_3 . . . 451
15.99 so_34907643_34908242_1 . . . 451
15.100 so_22137693_22137693_0 . . . 451
15.101 so_29182057_29182379_0 . . . 451
15.102 so_20397165_20397398_2 . . . 451
15.103 so_4094996_4095666_0 . . . 451
15.104 so_28588671_28588809_1 . . . 451
15.105 so_40991687_40991805_1 . . . 451
15.106 so_2372291_2372354_0 . . . 451
15.107 so_30821651_30821814_0 . . . 451
15.108 so_4508533_4508687_0 . . . 451
15.109 so_16613688_16613816_0 . . . 451
15.110 so_7333730_7333816_1 . . . 451
15.111 so_37273501_37274111_1 . . . 451
15.112 so_34007190_34007608_1 . . . 451
15.113 so_6864011_6864139_0 . . . 451
15.114 so_43609186_43609303_2 . . . 451
15.115 so_26214820_26214820_0 . . . 451
15.116 so_35858184_35858302_1 . . . 451
15.117 so_33115803_33115832_0 . . . 451
15.118 so_33115803_33161312_0 . . . 451
15.119 so_25094103_25094407_1 . . . 451
15.120 so_8311457_8311472_2 . . . 451
15.121 so_21691213_21691319_1 . . . 451

15.122 so_21691213_21691319_0 . . . 451
15.123 so_19657590_19658180_0 . . . 451
15.124 so_5952837_5955153_1 . . . 451
15.125 so_22137693_22137693_1 . . . 451
15.126 so_43565457_43565560_1 . . . 451
15.127 so_33689053_33689626_1 . . . 452
15.128 so_20397165_20397398_3 . . . 452
15.129 so_23011731_23013769_0 . . . 452
15.130 so_28588671_28588809_2 . . . 452
15.131 so_130117_739424_0 . . . 452
15.132 so_6506836_6506865_0 . . . 452
15.133 so_1886491_1886554_0 . . . 452
15.134 so_15222286_15222430_2 . . . 452
15.135 so_19603927_19604373_0 . . . 452
15.136 so_8311457_8311472_3 . . . 452
15.137 so_22135946_22136015_1 . . . 452
15.138 so_15293476_15293597_1 . . . 452
15.139 so_5798107_5798121_0 . . . 452
15.140 so_26395476_26395492_1 . . . 452
15.141 so_22137693_22137693_2 . . . 452
15.142 so_22137693_22137878_0 . . . 452
15.143 so_37788282_37788488_2 . . . 452
15.144 so_29182057_29182379_1 . . . 452
15.145 so_1482908_1482933_0 . . . 452
15.146 Handling an exception . . . 452
15.147 so_7298872_7298926_0 . . . 452
15.148 so_7356603_7356662_0 . . . 452
15.149 so_3856445_3860676_1 . . . 452
15.150 so_37881240_37881240_0 . . . 452
15.151 so_46891778_46891919_0 . . . 452
15.152 so_7991666_7991919_12 . . . 452
15.153 so_7991666_7991919_14 . . . 452
15.154 so_43892186_43892655_1 . . . 452
15.155 so_36168030_36168746_1 . . . 452
15.156 so_31526865_31527020_2 . . . 452
15.157 so_14526942_14587960_3 . . . 452
15.158 so_14526942_14587960_1 . . . 452
15.159 so_7991666_7991832_3 . . . 453
15.160 so_7991666_7991919_16 . . . 453
15.161 so_28974999_28975292_1 . . . 453
15.162 so_41533111_41533197_0 . . . 453
15.163 so_36168030_36168746_0 . . . 453
15.164 so_39527963_39537680_0 . . . 453
15.165 so_3605296_3605601_1 . . . 453
15.166 so_28974999_28975292_3 . . . 453
15.167 so_32890507_32890966_2 . . . 453
15.168 so_25638907_25640101_5 . . . 453
15.169 so_8595087_8859542_0 . . . 453
15.170 so_28267484_28267655_2 . . . 453
15.171 so_4245320_40158380_0 . . . 453
15.172 so_8597039_8597527_1 . . . 453

15.173 so_25638907_25640101_6 . . . 453
15.174 so_8595087_8859542_1 . . . 453
15.175 so_28267484_28267655_3 . . . 453
15.176 so_8597039_8597527_2 . . . 453
15.177 so_8553357_8554056_0 . . . 453
15.178 so_2633193_2633208_0 . . . 453
15.179 so_45995708_45996000_0 . . . 453
15.180 so_11063906_11063937_0 . . . 453
15.181 so_8144234_8144268_0 . . . 453
15.182 so_12887495_12887534_1 . . . 453
15.183 so_13982985_13983221_1 . . . 453
15.184 so_5557555_5557651_0 . . . 453
15.185 so_44887319_44887319_0 . . . 453
15.186 so_16870789_16870871_0 . . . 453
15.187 so_39962999_39963032_1 . . . 453
15.188 so_3570119_3571648_2 . . . 453
15.189 so_29182057_29182379_2 . . . 453
15.190 so_36724788_36724864_0 . . . 453
15.191 so_8367459_8367533_0 . . . 453
15.192 so_25094103_25094103_0 . . . 453
15.193 so_7360752_11814437_0 . . . 453
15.194 so_25094103_25094407_3 . . . 453
15.195 so_3181040_3181233_0 . . . 453
15.196 so_5070057_5070348_0 . . . 453
15.197 so_8405390_8405690_0 . . . 453
15.198 so_40377546_40377626_3 . . . 453
15.199 so_6756931_6756957_1 . . . 454
15.200 so_5255777_5255839_5 . . . 454
15.201 so_36273662_36273770_0 . . . 454
15.202 so_34370748_34371136_1 . . . 454
15.203 so_20614282_20614325_2 . . . 454
15.204 so_17564037_17564190_0 . . . 454
15.205 so_27011854_27011854_1 . . . 454
15.206 so_26267331_26267459_1 . . . 454
15.207 so_39118324_39118529_2 . . . 454
15.208 so_8405390_8405690_1 . . . 454
15.209 so_33015748_33016799_0 . . . 454
15.210 so_42859326_42859326_0 . . . 454
15.211 so_2743410_2743497_1 . . . 454
15.212 so_6756931_6756957_2 . . . 454
15.213 so_11574995_11575027_0 . . . 454
15.214 so_3181040_3181233_1 . . . 454
15.215 so_40499837_40499861_1 . . . 454
15.216 so_5952837_5952872_1 . . . 454
15.217 so_5070057_5070348_1 . . . 454
15.218 so_40537750_40538005_1 . . . 454
15.219 so_5401062_5401062_3 . . . 454
15.220 so_36273662_36273770_1 . . . 454
15.221 so_7401269_7404946_0 . . . 454
15.222 so_36168799_36168890_0 . . . 454
15.223 so_31526865_31527020_3 . . . 454

15.224 so_26883615_26883787_0 . . . 454
15.225 so_7401269_7404946_1 . . . 454
15.226 so_8481147_8481171_0 . . . 454
15.227 so_8481147_8481171_1 . . . 454
15.228 Exception specifications . . . 454
15.229 so_8576509_8576705_3 . . . 454
15.230 so_35551197_35553018_0 . . . 454
15.231 so_27488862_27489302_0 . . . 454
15.232 so_27489558_27489923_0 . . . 454
15.233 so_28974316_28974609_0 . . . 454
15.234 so_7903768_7905195_0 . . . 454
15.235 so_26838452_26838679_0 . . . 454
15.236 so_5472801_5473209_0 . . . 454
15.237 so_20610658_20616637_5 . . . 455
15.238 so_20610658_20616637_4 . . . 455
15.239 so_20610658_20616637_3 . . . 455
15.240 so_29776122_29776705_0 . . . 455
15.241 so_29850923_29854565_2 . . . 455
15.242 so_46946506_46946798_0 . . . 455
15.243 so_5295134_5295168_0 . . . 455
15.244 so_27489558_27489923_1 . . . 455
15.245 so_9635464_9635597_2 . . . 455
15.246 so_9635464_9635597_0 . . . 455
15.247 so_9635464_9635739_0 . . . 455
15.248 so_6864011_6864139_1 . . . 455
15.249 so_3233078_3233078_0 . . . 455
15.250 so_22204334_22204487_2 . . . 455
15.251 so_22204334_22204487_1 . . . 455
15.252 so_14859579_14860231_0 . . . 455
15.253 so_13125454_13125861_1 . . . 455
15.254 so_5472801_5474713_1 . . . 455
15.255 so_40226269_40226380_1 . . . 455
15.256 so_10227317_10227425_1 . . . 456
15.257 so_7060930_7060959_0 . . . 456
15.258 so_8576509_8576705_4 . . . 456
15.259 so_31809347_31809555_0 . . . 456
15.260 so_47583864_47584093_4 . . . 456
15.261 so_28974999_28975292_5 . . . 456
15.262 so_15449629_15449692_17 . . . 456
15.263 so_15449629_15449689_1 . . . 456
15.264 so_15449629_15449692_16 . . . 456
15.265 so_15449629_15449692_18 . . . 456
15.266 so_36295738_37138998_1 . . . 456
15.267 so_40226269_40233006_2 . . . 456
15.268 so_40226269_40233006_1 . . . 456
15.269 so_40226269_40233006_0 . . . 456
15.270 so_36225823_36226546_1 . . . 456
15.271 so_36225823_36226546_0 . . . 456
15.272 so_36225823_36226546_2 . . . 456
15.273 so_7312743_7312775_0 . . . 456
15.274 so_28974999_28975065_0 . . . 456

15.275 so_11497252_11497781_0 . . . 457
15.276 so_13041715_13044052_0 . . . 457
15.277 so_5472801_5474713_2 . . . 457
15.278 so_16244313_16244397_0 . . . 457
15.279 so_43363620_43364934_0 . . . 457
15.280 so_36760797_36760926_1 . . . 457
15.281 so_11497252_11497781_1 . . . 457
15.282 so_35039927_35039927_2 . . . 457
15.283 so_21214653_21214890_2 . . . 457
15.284 so_20610658_20616637_6 . . . 457
15.285 so_29623652_29623652_0 . . . 457
15.286 so_9180164_9180224_0 . . . 457
15.287 so_10756354_10756554_1 . . . 457
15.288 so_10731131_10731415_1 . . . 457
15.289 so_10731131_10731325_0 . . . 457
15.290 so_16540725_16540757_2 . . . 457
15.291 so_9635464_9635739_1 . . . 457
15.292 so_3233078_3233081_1 . . . 457
15.293 so_3233078_3233081_0 . . . 457
15.294 so_3233078_3233081_2 . . . 457
15.295 so_18416780_18416814_1 . . . 457
15.296 so_28046126_28046305_0 . . . 457
15.297 so_33057562_33057646_1 . . . 457
15.298 so_13041715_13041715_0 . . . 457
15.299 so_26331364_26332289_1 . . . 457
15.300 so_27737492_27737913_1 . . . 457
15.301 so_17177386_17178402_1 . . . 457
15.302 so_17177386_17177386_1 . . . 457
15.303 so_35991071_35991282_0 . . . 457
15.304 so_29850923_29854565_1 . . . 457
15.305 so_34581372_34581372_1 . . . 457
15.306 so_26395476_26395492_3 . . . 457
15.307 so_8576509_8576705_5 . . . 458
15.308 Special functions . . . 458
15.309 so_10091082_10091172_2 . . . 458
15.310 so_10091082_10091172_0 . . . 458
15.311 so_40656436_40656436_0 . . . 458
15.312 so_8237909_8237989_1 . . . 458
15.313 so_8237909_8237989_0 . . . 458
15.314 so_31602616_31602999_0 . . . 458
15.315 so_6185957_6186019_0 . . . 458
15.316 so_30250934_30251056_0 . . . 458
15.317 so_30250934_30251056_2 . . . 458
15.318 so_30250934_30251056_6 . . . 458
15.319 so_15449629_15449692_12 . . . 458
15.320 so_15449629_15449692_14 . . . 458
15.321 so_15449629_15449692_1 . . . 458
15.322 so_15449629_15449692_10 . . . 458
15.323 so_15449629_15449692_9 . . . 458
15.324 so_15449629_15449692_5 . . . 458
15.325 so_15449629_15449692_3 . . . 458

15.326 so_15449629_15449692_2 . . . 458
15.327 so_15449629_15449692_6 . . . 458
15.328 so_15449629_15449692_11 . . . 458
15.329 so_15449629_15449692_7 . . . 458
15.330 so_15449629_15449692_4 . . . 458
15.331 so_15449629_15449692_8 . . . 458
15.332 so_15449629_15449692_13 . . . 458
15.333 so_6537252_6537291_0 . . . 458
15.334 so_6537252_6537252_0 . . . 458
15.335 so_22438845_22438938_1 . . . 458
15.336 so_22438845_22438938_3 . . . 458
15.337 so_5557555_5557651_1 . . . 458
15.338 so_32890507_32890966_0 . . . 458
15.339 so_8065976_8066032_1 . . . 458
15.340 so_32323406_32323458_1 . . . 458
15.341 so_45497684_46754122_1 . . . 458
15.342 so_7319810_7319810_0 . . . 458
15.343 so_7319810_7320054_0 . . . 458
15.344 so_7319810_7319975_0 . . . 458
15.345 so_9971782_9972063_10 . . . 458
15.346 so_2438334_2438384_2 . . . 458
15.347 so_2438334_2438384_3 . . . 458
15.348 so_19344955_19345096_1 . . . 458
15.349 so_19517423_19517489_0 . . . 458
15.350 so_33689053_33689626_2 . . . 459
15.351 so_20397165_20397398_5 . . . 459
15.352 so_8553357_8554056_4 . . . 459
15.353 so_10091082_10091172_4 . . . 459
15.354 so_31602616_31602999_1 . . . 459
15.355 so_30250934_30251056_4 . . . 459
15.356 so_34481404_34481418_0 . . . 459
15.357 so_45220063_45220505_1 . . . 459
15.358 so_15449629_15449692_15 . . . 459
15.359 so_5557555_5557651_2 . . . 459
15.360 so_32890507_32890966_1 . . . 459
15.361 so_32890507_32890888_0 . . . 459
15.362 so_32323406_32323458_3 . . . 459
15.363 so_15798549_15801086_0 . . . 459
15.364 so_8311457_8311472_5 . . . 459
15.365 so_9971782_9972063_12 . . . 459
15.366 so_2438334_2438384_4 . . . 459
15.367 so_22137693_29576874_5 . . . 459
15.368 so_22137693_22137693_3 . . . 459
15.369 so_22137693_29576874_1 . . . 459
15.370 so_43363620_43364934_1 . . . 459
15.371 so_10056909_10056980_1 . . . 459
15.372 so_39962999_39962999_0 . . . 459
15.373 so_8597039_8597302_3 . . . 459

16 Preprocessing directives **460**
16.1 so_18682234_18682255_2 . . . 460

16.2 so_1262063_1262211_0 . . . 460
16.3 so_19321937_19321962_0 . . . 460
16.4 so_11912861_11912888_0 . . . 460
16.5 so_37622767_37623408_2 . . . 460
16.6 so_14763892_14764009_0 . . . 460
16.7 so_13754778_13769502_9 . . . 460
16.8 so_13754778_13769502_6 . . . 460
16.9 so_13754778_13758060_4 . . . 460
16.10 so_24988365_24988428_0 . . . 460
16.11 so_18474622_18474985_0 . . . 461
16.12 so_32838453_32838625_0 . . . 461
16.13 so_39737956_39740834_0 . . . 461
16.14 so_17213327_17213933_3 . . . 461
16.15 so_17213327_17213933_1 . . . 461
16.16 so_1262063_1262211_1 . . . 461
16.17 so_1262063_1262211_3 . . . 461
16.18 Conditional inclusion . . . 462
16.19 so_24037595_24037737_2 . . . 462
16.20 so_28199162_28199607_0 . . . 462
16.21 so_27278596_27278926_0 . . . 462
16.22 so_41773080_41773209_1 . . . 462
16.23 so_38355289_38355718_0 . . . 462
16.24 so_24037595_24037737_1 . . . 462
16.25 so_41455265_41455322_1 . . . 462
16.26 so_34018046_34018666_1 . . . 462
16.27 so_34018046_34018666_0 . . . 462
16.28 so_15598698_15598759_3 . . . 462
16.29 so_23986834_23986883_0 . . . 462
16.30 so_34108878_34108878_1 . . . 462
16.31 so_34677148_34679077_0 . . . 462
16.32 so_34555793_34556095_1 . . . 462
16.33 so_40570626_40570691_1 . . . 462
16.34 so_4219286_4219334_0 . . . 462
16.35 so_5085392_5085425_1 . . . 462
16.36 so_5085392_5085425_0 . . . 462
16.37 so_40928785_40928785_0 . . . 462
16.38 so_27278596_27278926_1 . . . 462
16.39 so_15208332_15208332_0 . . . 462
16.40 so_9722632_9723093_0 . . . 462
16.41 so_41773080_41773209_0 . . . 462
16.42 so_42575122_42575122_0 . . . 462
16.43 so_18903113_18903747_0 . . . 462
16.44 so_37619078_37619231_0 . . . 462
16.45 so_33916113_33916186_8 . . . 462
16.46 so_33916113_33923241_1 . . . 462
16.47 so_15935073_15935073_2 . . . 463
16.48 Source file inclusion . . . 463
16.49 so_1367316_1368711_2 . . . 463
16.50 so_17095966_17096009_0 . . . 463
16.51 so_3469588_3469588_2 . . . 463
16.52 so_24240064_24241265_0 . . . 463

16.53 so_1367316_1368711_6 . . . 463
16.54 so_30474867_30475447_2 . . . 463
16.55 so_38388947_38394333_0 . . . 463
16.56 so_40322496_40323377_2 . . . 463
16.57 so_42987150_42991635_4 . . . 463
16.58 so_26976962_26977312_0 . . . 463
16.59 so_5790161_5790245_0 . . . 463
16.60 so_21593_32064944_1 . . . 463
16.61 so_21593_41538_2 . . . 463
16.62 so_21593_77092_1 . . . 463
16.63 so_2123699_2123904_0 . . . 463
16.64 so_3469588_3469588_3 . . . 463
16.65 so_1951951_37228442_1 . . . 463
16.66 so_24240064_24241265_2 . . . 463
16.67 so_179213_1251308_1 . . . 463
16.68 so_3162030_3162067_1 . . . 463
16.69 so_10304751_10304797_2 . . . 463
16.70 so_1367316_1368711_11 . . . 463
16.71 so_1367316_1368711_9 . . . 463
16.72 so_30474867_30475447_7 . . . 463
16.73 so_30474867_30475447_5 . . . 463
16.74 so_38388947_38394333_1 . . . 463
16.75 so_40354900_40354998_2 . . . 463
16.76 so_42987150_42991635_9 . . . 463
16.77 so_42987150_42991635_7 . . . 463
16.78 so_21593_41538_7 . . . 463
16.79 so_21593_77092_4 . . . 463
16.80 so_21593_77092_3 . . . 463
16.81 so_21593_32064944_4 . . . 463
16.82 so_21593_32064944_3 . . . 463
16.83 so_21593_41538_5 . . . 463
16.84 so_34692366_34692862_3 . . . 463
16.85 so_40183851_40183851_0 . . . 463
16.86 so_3469588_3469588_4 . . . 463
16.87 so_1951951_37228442_2 . . . 463
16.88 so_5708259_26127812_1 . . . 463
16.89 so_24240064_24241265_5 . . . 463
16.90 so_24240064_24241265_4 . . . 463
16.91 so_179213_179436_4 . . . 463
16.92 so_179213_179436_2 . . . 463
16.93 so_179213_1251308_3 . . . 463
16.94 so_179213_1251308_4 . . . 463
16.95 so_21521333_21521556_2 . . . 463
16.96 so_3162030_3162067_4 . . . 463
16.97 so_3162030_3162067_3 . . . 463
16.98 so_10304751_10304797_7 . . . 463
16.99 so_10304751_10304797_5 . . . 463
16.100 so_13485192_13485406_0 . . . 463
16.101 so_8782343_8782718_0 . . . 463
16.102 so_1736654_1736711_1 . . . 463
16.103 so_29787673_29788014_5 . . . 463

16.104 so_21593_77092_6 . . . 463
16.105 so_21593_32064944_6 . . . 463
16.106 so_39038867_39074984_1 . . . 463
16.107 so_1262063_1262211_5 . . . 463
16.108 so_29720288_29720461_2 . . . 463
16.109 so_42858048_42858714_2 . . . 463
16.110 so_42858048_42858417_3 . . . 463
16.111 so_15014593_15014593_0 . . . 464
16.112 so_15731605_15731797_1 . . . 464
16.113 so_36290995_36292219_0 . . . 464
16.114 so_36290995_36292219_1 . . . 464
16.115 so_14909997_14909999_2 . . . 464
16.116 Macro replacement . . . 464
16.117 so_46731466_46736909_0 . . . 464
16.118 so_40215815_40216214_0 . . . 464
16.119 so_13384182_13384182_1 . . . 464
16.120 so_20523772_20525159_0 . . . 464
16.121 so_9349798_9349950_0 . . . 464
16.122 so_35210637_35212530_1 . . . 464
16.123 so_46794072_46794610_0 . . . 464
16.124 so_21266380_21266771_0 . . . 464
16.125 so_3301152_3301222_0 . . . 464
16.126 so_7666344_7674214_0 . . . 464
16.127 so_43028532_43061457_1 . . . 464
16.128 so_45456842_45457872_0 . . . 464
16.129 so_12131012_12131067_0 . . . 464
16.130 so_16767901_16769188_0 . . . 464
16.131 so_16767901_16767901_0 . . . 464
16.132 so_1736654_1736711_2 . . . 465
16.133 so_20889460_38163793_0 . . . 465
16.134 so_47549362_47549495_1 . . . 465
16.135 so_35210637_35212530_0 . . . 465
16.136 so_23022349_23022366_0 . . . 465
16.137 so_32241581_32262498_0 . . . 465
16.138 so_35210637_35212530_2 . . . 465
16.139 so_35210637_35212445_0 . . . 465
16.140 so_28343501_28343676_0 . . . 465
16.141 so_6502173_6502494_0 . . . 465
16.142 so_39534609_39534929_0 . . . 465
16.143 so_7666344_7673994_0 . . . 465
16.144 so_8101868_8101886_0 . . . 465
16.145 so_21139019_21139234_0 . . . 465
16.146 so_45456842_45457872_1 . . . 465
16.147 so_26049998_26050164_0 . . . 466
16.148 so_35210637_35212445_1 . . . 466
16.149 so_35210637_35212530_3 . . . 466
16.150 so_11106023_11106452_0 . . . 466
16.151 so_9003425_9003503_0 . . . 466
16.152 so_35710536_35711462_0 . . . 466
16.153 so_8231966_8232426_1 . . . 466
16.154 so_2949587_2949587_0 . . . 466

16.155 so_15889867_15889913_1 . . . 466
16.156 so_7666344_7673994_1 . . . 466
16.157 so_43028532_43061457_2 . . . 466
16.158 so_43028532_43061457_3 . . . 466
16.159 so_29794514_29795045_1 . . . 466
16.160 so_21010506_21010552_0 . . . 466
16.161 so_24506239_24506512_1 . . . 466
16.162 so_6463014_6463014_3 . . . 466
16.163 so_5354594_5354685_0 . . . 466
16.164 so_21098486_21100313_3 . . . 466
16.165 so_15813975_15814019_0 . . . 466
16.166 so_26219063_26724701_0 . . . 466
16.167 so_18513243_18513426_1 . . . 466
16.168 so_7666344_7674214_1 . . . 466
16.169 so_44088682_44091763_0 . . . 466
16.170 so_18379488_18380021_1 . . . 466
16.171 so_22666697_22668909_0 . . . 466
16.172 so_14763892_14764009_1 . . . 466
16.173 so_18759947_18760039_2 . . . 466
16.174 so_24506239_24506512_0 . . . 466
16.175 so_24506239_24506507_0 . . . 466
16.176 so_1121971_1122066_1 . . . 466
16.177 so_18682234_18682255_5 . . . 466
16.178 so_1121971_1122066_2 . . . 466
16.179 so_15889867_15889913_0 . . . 466
16.180 so_30426455_30447077_0 . . . 466
16.181 so_30426455_30447077_1 . . . 466
16.182 so_31433101_31433101_0 . . . 466
16.183 so_31433101_31435835_0 . . . 466
16.184 so_23218925_23221448_0 . . . 466
16.185 so_4892652_4892763_0 . . . 466
16.186 so_1121971_1122066_3 . . . 466
16.187 so_4225176_4225183_0 . . . 466
16.188 so_22806882_22807269_0 . . . 467
16.189 so_35210637_35212530_4 . . . 467
16.190 so_35210637_35212445_2 . . . 467
16.191 so_15583021_15583021_0 . . . 467
16.192 so_2949587_2949587_1 . . . 467
16.193 so_22590006_22590451_2 . . . 467
16.194 so_4892652_4892763_1 . . . 467
16.195 so_43028532_43061457_4 . . . 467
16.196 so_27519315_27519691_0 . . . 467
16.197 so_24405223_24405910_0 . . . 467
16.198 so_29183016_29183186_0 . . . 467
16.199 so_15583021_15583021_3 . . . 467
16.200 so_15583021_15583021_1 . . . 467
16.201 so_15583021_15583021_2 . . . 467
16.202 so_34504226_34504304_0 . . . 467
16.203 so_22502324_22502324_1 . . . 467
16.204 so_2949587_2949587_2 . . . 467
16.205 so_2949587_2949655_0 . . . 467

16.206 so_20889460_38163793_2 . . . 467
16.207 so_12262754_12263618_0 . . . 467
16.208 so_12447557_12447678_0 . . . 467
16.209 so_12447557_12447739_0 . . . 467
16.210 so_24831852_24832296_2 . . . 467
16.211 so_4670597_4670625_0 . . . 467
16.212 so_6598894_6599092_1 . . . 467
16.213 so_6598894_6599092_0 . . . 467
16.214 so_20889460_38163793_3 . . . 467
16.215 so_1262063_1262077_0 . . . 467
16.216 so_37622767_37623408_1 . . . 467
16.217 so_33388095_33388157_2 . . . 467
16.218 so_1135822_8289287_0 . . . 467
16.219 so_13895989_13896077_0 . . . 467
16.220 so_3314032_3315074_0 . . . 467
16.221 so_9090536_9090656_1 . . . 467
16.222 Line control . . . 470
16.223 so_9152854_9152887_1 . . . 470
16.224 so_9152854_9152887_3 . . . 470
16.225 so_9152854_9152887_5 . . . 470
16.226 Error directive . . . 470
16.227 so_24233566_24233609_2 . . . 470
16.228 Pragma directive . . . 471
16.229 so_17511269_17511426_0 . . . 471
16.230 so_17511269_17511382_1 . . . 471
16.231 so_13267648_13267797_5 . . . 471
16.232 so_13267648_13267797_2 . . . 471
16.233 so_26408426_26411022_0 . . . 471
16.234 so_23696115_23696175_1 . . . 471
16.235 so_26574435_26574610_2 . . . 471
16.236 so_12742725_12742843_1 . . . 471
16.237 Null directive . . . 471
16.238 Predefined macro names . . . 471
16.239 so_47549362_47549362_0 . . . 471
16.240 so_31430765_31430765_3 . . . 471
16.241 so_31430765_31430765_5 . . . 471
16.242 so_31430765_31430765_1 . . . 471
16.243 so_9889955_9890208_6 . . . 471
16.244 so_9889955_9890208_4 . . . 471
16.245 so_9889955_9890208_8 . . . 471
16.246 so_9889955_9890208_2 . . . 471
16.247 so_9889955_9890208_1 . . . 471
16.248 so_9889955_9890208_3 . . . 471
16.249 so_17127094_17127227_1 . . . 471
16.250 so_17127094_17127227_3 . . . 471
16.251 so_17127094_17127227_2 . . . 471
16.252 so_7530047_7530047_0 . . . 471
16.253 so_38456127_38456243_1 . . . 471
16.254 so_38456127_38456243_2 . . . 471
16.255 so_38456127_38456243_3 . . . 471
16.256 so_38593014_38594108_2 . . . 471

16.257 so_13777712_13777712_1 . . . 471
16.258 so_5075928_5076090_0 . . . 471
16.259 so_5075928_5076083_0 . . . 471
16.260 so_26089319_26089678_2 . . . 471
16.261 so_26089319_26089678_0 . . . 471
16.262 so_5047971_29407260_0 . . . 471
16.263 so_24622970_24635826_0 . . . 471
16.264 so_24622970_24635826_1 . . . 471
16.265 so_17292873_17292908_1 . . . 471
16.266 so_21946447_21946709_1 . . . 471
16.267 so_26534657_26534771_0 . . . 471
16.268 so_7755202_7755280_0 . . . 471
16.269 so_3768363_4248640_0 . . . 471
16.270 so_12590255_12590458_0 . . . 471
16.271 so_12590255_12590255_0 . . . 471
16.272 so_39548465_39548465_2 . . . 471
16.273 so_40751875_40751902_0 . . . 471
16.274 so_38770680_38776036_0 . . . 471
16.275 so_20797979_20798081_0 . . . 471
16.276 Pragma operator . . . 472

17 Library introduction **474**
17.1 General . . . 474
17.2 so_29720288_29720461_3 . . . 474
17.3 The C standard library . . . 475
17.4 so_44241289_44241651_1 . . . 475
17.5 so_15656434_15656459_0 . . . 475
17.6 so_44760245_44760695_0 . . . 475
17.7 so_6434549_6434582_0 . . . 475
17.8 so_26304506_26304633_6 . . . 475
17.9 Definitions . . . 475
17.10 Additional definitions . . . 478
17.11 Method of description (Informative) . . . 478
17.12 so_21364398_21364398_0 . . . 478
17.13 so_42621378_46537596_2 . . . 479
17.14 so_5322104_5322234_3 . . . 479
17.15 so_14970747_14971184_1 . . . 481
17.16 so_31210366_31210815_0 . . . 481
17.17 so_26230791_26231031_0 . . . 481
17.18 so_15727869_15727869_0 . . . 481
17.19 so_15727869_15727869_1 . . . 481
17.20 so_15727869_15727869_2 . . . 481
17.21 so_27988870_27989236_1 . . . 482
17.22 so_12773246_12773395_0 . . . 482
17.23 so_13545005_13545005_0 . . . 482
17.24 so_34965820_34985675_1 . . . 482
17.25 so_4025458_4025458_1 . . . 483
17.26 so_4025458_4025458_2 . . . 483
17.27 so_27871124_27871124_0 . . . 483
17.28 so_24469809_24469850_12 . . . 483
17.29 so_9248421_9248495_0 . . . 484

17.30 so_13902910_13912596_0 . . . 484
17.31 so_14272141_14285236_0 . . . 484
17.32 so_17627302_17628398_1 . . . 484
17.33 so_26315899_26417168_0 . . . 484
17.34 so_34493104_34493170_0 . . . 484
17.35 so_17627302_17628398_2 . . . 484
17.36 so_13902910_13912596_2 . . . 484
17.37 Library-wide requirements . . . 484
17.38 so_21655523_21655523_1 . . . 484
17.39 so_25698780_25699475_0 . . . 484
17.40 so_25698780_25699475_1 . . . 484
17.41 so_41536696_41536809_0 . . . 484
17.42 so_17504565_17505654_0 . . . 484
17.43 so_19588081_19588547_0 . . . 484
17.44 so_15045651_15045715_2 . . . 484
17.45 so_28125290_28125378_0 . . . 484
17.46 so_35737741_35737897_1 . . . 484
17.47 so_42957970_42958577_0 . . . 484
17.48 so_1687860_1688142_0 . . . 484
17.49 so_16506196_16506229_0 . . . 485
17.50 so_3469588_3469588_5 . . . 485
17.51 so_27261508_27261508_1 . . . 485
17.52 so_6469954_6470072_0 . . . 485
17.53 so_38210174_38210199_0 . . . 485
17.54 so_16506196_16506229_1 . . . 485
17.55 so_2649207_2649315_0 . . . 485
17.56 so_16506196_16506229_2 . . . 485
17.57 so_19123631_19123761_0 . . . 485
17.58 so_10445586_10445657_1 . . . 485
17.59 so_10823878_10824486_0 . . . 485
17.60 so_29897661_29900474_0 . . . 485
17.61 so_2649207_2649315_1 . . . 485
17.62 so_17504565_17505654_1 . . . 485
17.63 so_10466096_10466096_0 . . . 485
17.64 so_12696764_12696998_1 . . . 485
17.65 so_34240531_34240548_0 . . . 485
17.66 so_11892976_11893270_0 . . . 485
17.67 so_11892976_11893237_0 . . . 485
17.68 so_11085916_11086087_0 . . . 485
17.69 so_2118422_2118535_1 . . . 485
17.70 so_32354282_32358211_1 . . . 485
17.71 so_41791272_41791540_0 . . . 485
17.72 so_9149368_9149374_2 . . . 485
17.73 so_29551755_29551811_0 . . . 485
17.74 so_7131858_7132065_5 . . . 485
17.75 so_26821576_26821653_0 . . . 485
17.76 so_15656434_15656459_1 . . . 485
17.77 so_16771879_16772016_1 . . . 485
17.78 so_26304506_26304633_7 . . . 485
17.79 so_13040139_13040241_0 . . . 485
17.80 so_12574492_12574492_0 . . . 485

17.81 so_1524139_1524387_0 . . . 485
17.82 so_2587445_2587445_0 . . . 485
17.83 so_43263880_43264471_0 . . . 485
17.84 so_13129945_13130023_1 . . . 485
17.85 so_13129945_13130023_0 . . . 485
17.86 so_8087438_8087599_0 . . . 485
17.87 so_13244540_13245532_0 . . . 485
17.88 so_37715537_37715537_0 . . . 485
17.89 so_37715537_37715594_0 . . . 485
17.90 so_37715537_37715681_0 . . . 485
17.91 so_15604129_15604270_0 . . . 485
17.92 so_44369648_44370539_0 . . . 486
17.93 so_20170578_20170763_0 . . . 486
17.94 so_7935913_7935957_0 . . . 486
17.95 so_44369648_44370539_1 . . . 486
17.96 so_7131858_7132065_3 . . . 486
17.97 so_44369648_44370539_4 . . . 486
17.98 so_204476_18721336_31 . . . 486
17.99 so_13517470_13517627_0 . . . 486
17.100 so_30271822_30275587_1 . . . 486
17.101 so_2776381_2776639_1 . . . 486
17.102 so_204476_18721336_32 . . . 486
17.103 so_13517470_13517627_1 . . . 486
17.104 so_30271822_30275587_2 . . . 486
17.105 so_7050925_7050940_2 . . . 486
17.106 so_2776381_2776639_2 . . . 486
17.107 so_204476_18721336_33 . . . 486
17.108 so_42441683_42442198_5 . . . 486
17.109 so_30271822_30275587_3 . . . 486
17.110 so_28747481_28747481_2 . . . 487
17.111 so_34236653_34236783_2 . . . 487
17.112 so_19738711_19738767_0 . . . 487
17.113 so_8483955_8484037_0 . . . 487
17.114 so_28747481_28747481_1 . . . 487
17.115 so_27749974_27750058_0 . . . 487
17.116 so_29338108_29338121_0 . . . 487
17.117 so_16506196_16506196_0 . . . 487
17.118 so_12324302_12324346_0 . . . 487
17.119 so_6841130_6841534_0 . . . 487
17.120 so_4601700_4601798_0 . . . 487
17.121 so_22115996_22116498_2 . . . 487
17.122 so_32259671_32259806_0 . . . 487
17.123 so_39961314_39961431_1 . . . 487
17.124 so_13678028_13678176_0 . . . 487
17.125 so_25633852_25634043_0 . . . 487
17.126 so_3571771_3571776_0 . . . 487
17.127 so_32285139_32309138_0 . . . 487
17.128 so_26160465_26160651_1 . . . 487
17.129 so_8654649_8655465_1 . . . 487
17.130 so_43606777_43607151_0 . . . 487
17.131 so_695645_695652_0 . . . 488

17.132 so_35444114_35445106_3 . . . 488
17.133 so_41140170_41149207_0 . . . 488
17.134 so_6221610_6221635_2 . . . 488
17.135 so_5215249_5215292_1 . . . 488
17.136 so_28205387_28205411_1 . . . 488
17.137 so_10396278_10396955_5 . . . 488
17.138 so_43606777_43607151_1 . . . 488
17.139 so_45014513_45014642_1 . . . 488
17.140 so_13127455_13127916_2 . . . 488
17.141 so_26579132_26579280_0 . . . 488
17.142 so_26579132_26579368_1 . . . 488
17.143 so_13129031_13129446_0 . . . 488
17.144 so_47021238_47021276_0 . . . 489
17.145 so_36850522_36851037_7 . . . 489
17.146 so_8219525_8219718_2 . . . 489
17.147 so_14402990_14403772_4 . . . 489
17.148 so_14402990_14403772_3 . . . 489
17.149 so_14024228_14500519_1 . . . 489
17.150 so_14024228_14500519_2 . . . 489
17.151 so_14024228_14500519_0 . . . 489
17.152 so_22915325_23657560_0 . . . 490
17.153 so_44356761_44411583_2 . . . 490
17.154 so_36850522_36851037_3 . . . 490
17.155 so_27646996_27646996_2 . . . 490
17.156 so_4114794_7378683_3 . . . 491
17.157 so_10723892_10724143_3 . . . 491
17.158 so_40088029_40088115_1 . . . 491
17.159 so_14063717_14067836_1 . . . 491
17.160 so_37720832_37917195_0 . . . 491
17.161 so_47203934_47204428_0 . . . 491
17.162 so_43956125_44221983_1 . . . 491
17.163 so_11990794_11990952_1 . . . 491
17.164 so_11990794_11990992_1 . . . 491
17.165 so_29632309_29632634_0 . . . 491
17.166 so_27830323_27831380_14 . . . 492
17.167 so_32602235_32988737_3 . . . 492
17.168 so_6065814_6065831_2 . . . 492
17.169 so_33038802_33039097_0 . . . 492
17.170 so_27878843_27883301_0 . . . 492
17.171 so_34610955_34610955_0 . . . 492
17.172 so_31936418_31936575_0 . . . 492
17.173 so_31837603_31856601_0 . . . 492
17.174 so_21644735_21644735_0 . . . 492
17.175 so_24595837_24597217_0 . . . 492
17.176 so_28142011_28142011_1 . . . 492
17.177 so_37750904_37751139_2 . . . 492
17.178 so_24803430_24846295_1 . . . 492
17.179 so_12498709_12499353_0 . . . 492
17.180 so_12498709_12499353_1 . . . 492
17.181 so_3802454_3802454_0 . . . 492
17.182 so_4773334_4773351_1 . . . 492

17.183 so_12909017_12910073_1 . . . 492
17.184 so_12362363_12362522_0 . . . 495
17.185 so_34861283_34863387_1 . . . 495
17.186 so_40315492_40315681_0 . . . 496
17.187 so_45132546_45132840_0 . . . 496
17.188 so_25806959_25806959_1 . . . 496
17.189 so_27646996_27646996_1 . . . 496
17.190 so_12498709_12499353_2 . . . 496
17.191 so_29285322_29285323_0 . . . 496
17.192 so_29262292_29262751_2 . . . 496
17.193 so_24278803_28995278_2 . . . 496
17.194 so_16425359_16429114_1 . . . 497
17.195 so_30311551_30311552_0 . . . 497
17.196 so_9109377_9110053_1 . . . 497
17.197 so_25893317_25893454_1 . . . 497
17.198 so_27778908_27779038_1 . . . 497
17.199 so_41303762_41304226_1 . . . 497
17.200 so_11562_109613_0 . . . 497
17.201 so_8135555_8136519_0 . . . 497
17.202 so_41062294_41062439_0 . . . 497
17.203 so_3699009_3699009_0 . . . 497
17.204 so_25120423_25120423_0 . . . 497
17.205 so_16122912_16519653_0 . . . 497
17.206 so_16122912_16519653_2 . . . 497
17.207 so_13240141_13240708_1 . . . 497
17.208 so_15188937_47136505_0 . . . 497
17.209 so_17136497_17136607_0 . . . 497
17.210 so_30020005_30020407_3 . . . 497
17.211 so_6885063_6885137_0 . . . 497
17.212 so_30177621_30177658_0 . . . 497
17.213 so_27396640_27396905_1 . . . 497
17.214 so_44390785_44390926_0 . . . 497
17.215 so_23659222_23659222_0 . . . 497
17.216 so_8836388_8837166_0 . . . 497
17.217 so_8513417_8513497_0 . . . 497
17.218 so_22176268_22180875_0 . . . 497
17.219 so_36594569_36596739_1 . . . 497
17.220 so_23870522_23870601_0 . . . 497
17.221 so_9576112_9576112_0 . . . 497
17.222 so_6313771_6313771_0 . . . 497
17.223 so_28077592_28077711_0 . . . 497
17.224 so_28077592_28077711_1 . . . 497
17.225 so_41999583_41999583_0 . . . 497
17.226 so_33399594_33399722_0 . . . 497
17.227 so_12732042_12732136_3 . . . 497
17.228 so_12732042_12732136_2 . . . 497
17.229 so_6575966_6576087_0 . . . 497
17.230 so_6575966_6576087_1 . . . 497
17.231 so_27137634_47136630_1 . . . 497
17.232 so_27137634_47136630_7 . . . 497
17.233 so_9646297_23311961_2 . . . 497

17.234 so_9646297_23311961_1 . . . 497
17.235 so_36813760_36813965_5 . . . 497
17.236 so_25169745_25169918_0 . . . 497
17.237 so_28447236_47223308_0 . . . 497
17.238 so_2223245_14769750_0 . . . 497
17.239 so_37306477_37308265_1 . . . 497
17.240 so_34307129_34311044_8 . . . 497
17.241 so_34307129_34311044_2 . . . 497
17.242 so_10289766_10290176_0 . . . 497
17.243 so_19325984_19327299_0 . . . 497
17.244 so_15486928_15486996_2 . . . 497
17.245 so_14402990_14403772_1 . . . 497
17.246 so_16083926_16083983_0 . . . 497
17.247 so_46625283_46625423_0 . . . 497
17.248 so_35737741_35737897_2 . . . 497
17.249 so_3623631_3623647_1 . . . 497
17.250 so_25992729_25992877_0 . . . 497
17.251 so_23339298_23339498_0 . . . 497
17.252 so_40584368_40584504_0 . . . 497
17.253 so_36533199_36533280_1 . . . 497
17.254 so_36533199_36533280_2 . . . 497
17.255 so_38207659_38208127_2 . . . 497
17.256 so_38207659_38207659_1 . . . 497
17.257 so_320798_9149437_0 . . . 497
17.258 so_37541022_37541022_0 . . . 497
17.259 so_20590656_20602159_0 . . . 497
17.260 so_47029990_47030147_0 . . . 497
17.261 so_34629249_34629426_0 . . . 497
17.262 so_34629249_34629398_0 . . . 497
17.263 so_31473756_31473875_0 . . . 497
17.264 so_27630106_27630415_0 . . . 497
17.265 so_19081133_19134895_0 . . . 497
17.266 so_27206727_27207972_0 . . . 497
17.267 so_22701354_22701397_0 . . . 497
17.268 so_10736046_10736099_0 . . . 497
17.269 so_10736046_10736392_0 . . . 497
17.270 so_16366674_16367158_0 . . . 497
17.271 so_41062294_41062439_3 . . . 497
17.272 so_41062294_41062439_5 . . . 497
17.273 so_41062294_41062439_4 . . . 497
17.274 so_41062294_41062439_2 . . . 497
17.275 so_41915587_41916709_1 . . . 497
17.276 so_16122912_16519653_1 . . . 497
17.277 so_8513417_8513497_3 . . . 497
17.278 so_8513417_8513497_4 . . . 497
17.279 so_8513417_8513497_5 . . . 497
17.280 so_8513417_8513497_2 . . . 497
17.281 so_6575966_6576087_3 . . . 497
17.282 so_6575966_6576087_4 . . . 497
17.283 so_34307129_34311044_4 . . . 497
17.284 so_34307129_34311044_10 . . . 497

17.285 so_34307129_34311044_6 . . . 497
17.286 so_34307129_34311044_11 . . . 497
17.287 so_34307129_34311044_5 . . . 497
17.288 so_34307129_34311044_12 . . . 497
17.289 so_27972087_27972412_0 . . . 497
17.290 so_23339298_23339298_2 . . . 497
17.291 so_36533199_36533280_8 . . . 497
17.292 so_36533199_36533280_9 . . . 497
17.293 so_36533199_36533280_7 . . . 497
17.294 so_36533199_36533280_6 . . . 497
17.295 so_38207659_38208127_1 . . . 497
17.296 so_41999583_41999583_1 . . . 497
17.297 so_43263880_43264411_0 . . . 498
17.298 so_44580584_44580806_2 . . . 498
17.299 so_34410668_34411525_0 . . . 498
17.300 so_17504565_17504565_4 . . . 498
17.301 so_32285139_32309138_1 . . . 498
17.302 so_19790804_19791281_0 . . . 498
17.303 so_41746617_41785718_1 . . . 498
17.304 so_22489960_22490334_1 . . . 498
17.305 so_11892976_11893237_3 . . . 498
17.306 so_43884847_43885398_2 . . . 498
17.307 so_18449296_18449385_2 . . . 498
17.308 so_35134633_35134977_2 . . . 498
17.309 so_43263880_43264411_1 . . . 498
17.310 so_44580584_44580806_4 . . . 498
17.311 so_22586250_22586378_2 . . . 498
17.312 so_37578352_37578390_0 . . . 498
17.313 so_6128655_6128711_1 . . . 498
17.314 so_5777991_5795857_0 . . . 498
17.315 so_45754194_45754633_1 . . . 498
17.316 so_45994355_45994543_0 . . . 498
17.317 so_31751723_31763916_1 . . . 498
17.318 so_44580584_44580806_5 . . . 498
17.319 so_21966230_21968228_3 . . . 498
17.320 so_30311551_30311552_1 . . . 498
17.321 so_40461250_40461383_0 . . . 498
17.322 so_22489960_22490334_0 . . . 498
17.323 so_22259199_22259253_0 . . . 498
17.324 so_27377407_27377407_0 . . . 498
17.325 so_25893317_25893454_0 . . . 498
17.326 so_12100789_12100931_0 . . . 498
17.327 so_26959188_26959277_1 . . . 498
17.328 so_45754194_45754633_2 . . . 498
17.329 so_27778908_27779609_0 . . . 498
17.330 so_27778908_27779038_0 . . . 498
17.331 so_45994355_45994543_1 . . . 498
17.332 so_2726204_2726221_1 . . . 498
17.333 so_17213327_17213933_7 . . . 498
17.334 so_27903704_27903704_0 . . . 498
17.335 so_22736150_22736198_3 . . . 498

17.336 so_22736150_22736198_1 . . . 498
17.337 so_22736150_22736198_2 . . . 498
17.338 so_27777071_27777306_2 . . . 498
17.339 so_9668947_9669116_1 . . . 498
17.340 so_9668947_9668947_0 . . . 498
17.341 so_45784116_45784116_0 . . . 498
17.342 so_1421694_1421810_0 . . . 498
17.343 so_2770355_2770987_0 . . . 498
17.344 so_37325293_37325293_1 . . . 498
17.345 so_33005579_33005611_2 . . . 498
17.346 so_33005579_33005611_1 . . . 498
17.347 so_33005579_33005607_0 . . . 498
17.348 so_19790804_19791281_1 . . . 498
17.349 so_7979797_7979846_1 . . . 498
17.350 so_7979797_7979846_0 . . . 498
17.351 so_7979797_7979846_2 . . . 498
17.352 so_33856187_33856325_1 . . . 498
17.353 so_33856187_33856325_2 . . . 498
17.354 so_33856187_33856325_3 . . . 498
17.355 so_33856187_33856407_2 . . . 498
17.356 so_12924243_12924268_1 . . . 498
17.357 so_12924243_12924268_2 . . . 498
17.358 so_12924243_12924268_0 . . . 498
17.359 so_4180166_4180207_2 . . . 498
17.360 so_4180166_4180207_1 . . . 498
17.361 so_4180166_4180207_0 . . . 498
17.362 so_7399397_7399575_1 . . . 498
17.363 so_7399397_7399575_2 . . . 498
17.364 so_12414804_12415039_0 . . . 498
17.365 so_29971760_29971760_1 . . . 498
17.366 so_29971760_29972267_0 . . . 498
17.367 so_8692013_8692013_0 . . . 498
17.368 so_3540841_3540883_0 . . . 498
17.369 so_32520257_32520507_2 . . . 498
17.370 so_32520257_32520507_0 . . . 498
17.371 so_32520257_32520507_1 . . . 498
17.372 so_40515408_40515434_1 . . . 498
17.373 so_44209259_44209886_2 . . . 498
17.374 so_44209259_44209886_1 . . . 498
17.375 so_44209259_44209886_3 . . . 498
17.376 so_1399215_1399292_0 . . . 498
17.377 so_40434664_40436485_0 . . . 498
17.378 so_5777991_5777991_0 . . . 498
17.379 so_12721584_12721584_1 . . . 498
17.380 so_22619447_22636581_2 . . . 498
17.381 so_22619447_22636581_1 . . . 498
17.382 so_22619447_22636581_3 . . . 498
17.383 so_6756059_6756290_0 . . . 498
17.384 so_314983_315020_0 . . . 498
17.385 so_47120549_47120582_0 . . . 498
17.386 so_33822954_33823014_0 . . . 498

17.387 so_28291195_28291232_0 . . . 498
17.388 so_35773773_35774127_0 . . . 498
17.389 so_22561847_22562263_0 . . . 498
17.390 so_10802493_10803273_1 . . . 498
17.391 so_228783_228848_3 . . . 498
17.392 so_228783_228848_2 . . . 498
17.393 so_228783_228848_1 . . . 498
17.394 so_228783_228797_0 . . . 498
17.395 so_22591490_22591837_3 . . . 498
17.396 so_22591490_22591837_4 . . . 498
17.397 so_15727103_15727264_0 . . . 498
17.398 so_36547516_36552661_5 . . . 498
17.399 so_36547516_36552661_6 . . . 498
17.400 so_4270316_4270347_0 . . . 498
17.401 so_4270316_4270347_1 . . . 498
17.402 so_4270316_4270339_2 . . . 498
17.403 so_4270316_4270339_3 . . . 498
17.404 so_4270316_4270347_2 . . . 498
17.405 so_15840058_15840099_0 . . . 498
17.406 so_13040139_13040560_0 . . . 498
17.407 so_17754324_17754750_0 . . . 498
17.408 so_41746617_41785718_0 . . . 498
17.409 so_11892976_11893237_1 . . . 498
17.410 so_8692013_8692035_1 . . . 498
17.411 so_43884847_43885398_0 . . . 498
17.412 so_26821576_26821655_2 . . . 498
17.413 so_43263880_43264471_1 . . . 498
17.414 so_31136022_31136204_0 . . . 498
17.415 so_19696782_19696833_2 . . . 498
17.416 so_17754324_17754750_1 . . . 498
17.417 so_11892976_11893237_2 . . . 498
17.418 so_43884847_43885398_1 . . . 498
17.419 so_26160465_26160651_0 . . . 498
17.420 so_19696782_19696833_3 . . . 498
17.421 so_10466096_10469611_1 . . . 499
17.422 so_5813700_5813804_0 . . . 499
17.423 so_5813700_5813804_1 . . . 499
17.424 so_19892137_19892338_0 . . . 499
17.425 so_17197000_17197000_0 . . . 499
17.426 so_12924243_12924268_3 . . . 499
17.427 so_32250408_32250549_2 . . . 499
17.428 so_228783_228848_5 . . . 499
17.429 so_24770119_24770416_2 . . . 499
17.430 so_36843514_36845264_1 . . . 499
17.431 so_29983869_29984358_1 . . . 499
17.432 so_619467_619590_3 . . . 499
17.433 so_42441683_42442198_6 . . . 499
17.434 so_36843514_36845264_2 . . . 500
17.435 so_11846511_11848142_0 . . . 500
17.436 so_24770119_24770416_3 . . . 500
17.437 so_15367060_15367324_1 . . . 500

17.438 so_15367060_15367324_0 . . . 500
17.439 so_704780_705611_6 . . . 500
17.440 so_26043871_26044156_0 . . . 500
17.441 so_43791221_43907314_0 . . . 500
17.442 so_29303939_29304084_0 . . . 500
17.443 so_35444114_35445106_4 . . . 500
17.444 so_12423460_12428096_0 . . . 500
17.445 so_29262292_29262751_0 . . . 500
17.446 so_12434412_12434578_0 . . . 500
17.447 so_700588_702111_0 . . . 500
17.448 so_45905122_45905608_2 . . . 500
17.449 so_4940419_4940550_0 . . . 500
17.450 so_28911009_28911018_0 . . . 500
17.451 so_13089388_13089641_0 . . . 500
17.452 so_16876410_16877779_1 . . . 500
17.453 so_704780_705611_7 . . . 500
17.454 so_7776809_13877731_0 . . . 500
17.455 so_26043871_26044156_1 . . . 500
17.456 so_43791221_43907314_3 . . . 500
17.457 so_43791221_43907314_1 . . . 500
17.458 so_26902006_26913128_3 . . . 500
17.459 so_26902006_26913128_4 . . . 500
17.460 so_26902006_26913128_1 . . . 500
17.461 so_26902006_26913128_5 . . . 500
17.462 so_26902006_26913128_2 . . . 500
17.463 so_26902006_26913128_0 . . . 500
17.464 so_29303939_29304084_1 . . . 500
17.465 so_30771765_30772158_0 . . . 500
17.466 so_26526046_26526106_1 . . . 500
17.467 so_12281274_12295799_2 . . . 500
17.468 so_12281274_12295799_0 . . . 500
17.469 so_40821950_40822131_3 . . . 500
17.470 so_8329826_8329826_0 . . . 500
17.471 so_40311194_40311194_0 . . . 500
17.472 so_10971456_10976746_1 . . . 500
17.473 so_10971456_10976746_0 . . . 500
17.474 so_35444114_35445106_5 . . . 500
17.475 so_12423460_12428096_1 . . . 500
17.476 so_12423460_12428096_3 . . . 500
17.477 so_12423460_12428096_5 . . . 500
17.478 so_10065384_11514677_0 . . . 500
17.479 so_34659923_34660570_0 . . . 500
17.480 so_34659923_34660681_0 . . . 500
17.481 so_6226626_6226667_0 . . . 500
17.482 so_33895811_33896020_0 . . . 500
17.483 so_29262292_29262751_1 . . . 500
17.484 so_35461561_35462560_0 . . . 500
17.485 so_13224919_13225127_4 . . . 500
17.486 so_13224919_13225127_3 . . . 500
17.487 so_13224919_13225127_1 . . . 500
17.488 so_14830112_14830481_2 . . . 500

17.489 so_17909386_17909531_0 . . . 500
17.490 so_31345193_31347287_0 . . . 500
17.491 so_31345193_31347287_3 . . . 500
17.492 so_17325473_17325730_0 . . . 500
17.493 so_12434412_12434578_1 . . . 500
17.494 so_6517231_6517325_1 . . . 500
17.495 so_6517231_6517325_0 . . . 500
17.496 so_700588_702111_1 . . . 500
17.497 so_37767893_37768220_0 . . . 500
17.498 so_15102111_15104537_0 . . . 500
17.499 so_38898935_38899219_0 . . . 500
17.500 so_4940419_4940550_1 . . . 500
17.501 so_28911009_28911018_1 . . . 500
17.502 so_37327531_37328501_0 . . . 500
17.503 so_31047457_31047542_0 . . . 500
17.504 so_7730211_7730244_0 . . . 500
17.505 so_21007724_21007899_0 . . . 500
17.506 so_21007724_21007793_0 . . . 500
17.507 so_13089388_13089641_1 . . . 500
17.508 so_37788282_37788488_1 . . . 500
17.509 so_16505782_16509511_4 . . . 500
17.510 so_7730874_7730874_0 . . . 500
17.511 so_34562865_34566193_1 . . . 500
17.512 so_34562865_34566193_0 . . . 500
17.513 so_20196383_20196749_0 . . . 500
17.514 so_24335886_24336049_0 . . . 501
17.515 so_24156369_24166309_0 . . . 501
17.516 so_16616253_16659850_0 . . . 501
17.517 so_24444630_24444630_0 . . . 501
17.518 so_18027177_18027177_0 . . . 501
17.519 so_14526901_14526944_2 . . . 501
17.520 so_28359555_28359582_0 . . . 501
17.521 so_39664672_39665291_0 . . . 501
17.522 so_17873575_17883966_0 . . . 501
17.523 so_25809873_25814421_0 . . . 501
17.524 so_25809873_25814424_0 . . . 501
17.525 so_29300408_29300863_0 . . . 501
17.526 so_10043716_10043883_0 . . . 501
17.527 so_19541426_19541708_2 . . . 501
17.528 so_19541426_19541904_1 . . . 501
17.529 so_30695810_30696007_0 . . . 501
17.530 so_31120621_31127551_2 . . . 501
17.531 so_24604285_24604504_0 . . . 501
17.532 so_9585990_9588075_3 . . . 501
17.533 so_42114080_42114080_0 . . . 501
17.534 so_13127455_13127916_0 . . . 501
17.535 so_13129031_13129031_0 . . . 501
17.536 so_17309708_17309820_0 . . . 501
17.537 so_9274250_9274327_0 . . . 501
17.538 so_9042571_9042626_0 . . . 501
17.539 so_27829806_27829863_0 . . . 501

17.540 so_17957366_17961043_3 . . . 501
17.541 so_25803403_25803813_0 . . . 501
17.542 so_31910794_31910836_6 . . . 501
17.543 so_22146749_22147471_4 . . . 501
17.544 so_29654728_29654925_1 . . . 501
17.545 so_27864276_27864309_0 . . . 502
17.546 so_34881205_34882387_0 . . . 502
17.547 so_37589226_37589226_0 . . . 502
17.548 so_32354282_32358211_0 . . . 502
17.549 so_7131858_7132065_7 . . . 502
17.550 so_32670531_32670656_0 . . . 502
17.551 so_29454488_29456602_0 . . . 502
17.552 so_38421334_38421743_2 . . . 502
17.553 so_28745478_28747623_0 . . . 502
17.554 so_28745478_28747623_1 . . . 502
17.555 so_31751723_31763916_0 . . . 502
17.556 so_28747481_28747481_3 . . . 502
17.557 so_29095324_29095439_0 . . . 502
17.558 so_32466566_32466855_0 . . . 502
17.559 so_27864276_27864309_1 . . . 502
17.560 so_1119370_1119390_2 . . . 502
17.561 so_27864276_27864309_2 . . . 502
17.562 so_28745478_28747623_2 . . . 502
17.563 so_28747481_28747791_1 . . . 502
17.564 so_37325293_37326614_0 . . . 502
17.565 so_33993032_33993212_0 . . . 502
17.566 so_34559542_34559575_0 . . . 502
17.567 so_13521030_13521268_1 . . . 502
17.568 so_8719829_8719829_1 . . . 502
17.569 so_34559542_34559575_1 . . . 503
17.570 so_30770071_30770449_0 . . . 503
17.571 so_30770071_30770449_1 . . . 503
17.572 so_13521030_13521268_0 . . . 503
17.573 so_5031856_5031972_2 . . . 503
17.574 so_5031856_5031972_4 . . . 503
17.575 so_5031856_5031972_3 . . . 503
17.576 so_5031856_5031972_1 . . . 503
17.577 so_29118101_29119264_0 . . . 503
17.578 so_6007563_6008660_2 . . . 503
17.579 so_14882588_14884118_1 . . . 503
17.580 so_14882588_14884118_2 . . . 503
17.581 so_21539291_21539291_0 . . . 503
17.582 so_21539291_21539291_3 . . . 503
17.583 so_21539291_21539291_1 . . . 503
17.584 so_21539291_21539291_2 . . . 503
17.585 so_41554125_41556825_0 . . . 503
17.586 so_30111320_30111560_2 . . . 503
17.587 so_30111320_30111560_1 . . . 503
17.588 so_23462297_23462466_1 . . . 503
17.589 so_23462297_23462466_2 . . . 503
17.590 so_38017567_38024739_0 . . . 503

17.591 so_29200036_29200718_0 . . . 503
17.592 so_29200036_29200718_1 . . . 503
17.593 so_7820981_7820981_1 . . . 503
17.594 so_7820981_7820981_0 . . . 503
17.595 so_5031856_5031972_5 . . . 503
17.596 so_21539291_21539291_4 . . . 503
17.597 so_21539291_21561407_0 . . . 503
17.598 so_41554125_41556825_2 . . . 503
17.599 so_36497632_36497920_1 . . . 503
17.600 so_27708629_29457433_7 . . . 503
17.601 so_22182432_27906899_2 . . . 503
17.602 so_27744079_27744080_1 . . . 503
17.603 so_609956_610146_1 . . . 503
17.604 so_34343619_34343710_0 . . . 503
17.605 so_20755194_22301994_0 . . . 503
17.606 so_46202087_46202508_0 . . . 503
17.607 so_22692783_22694959_0 . . . 503
17.608 so_21766477_21767867_1 . . . 504
17.609 so_10833512_10833512_4 . . . 504
17.610 so_19000992_19001141_1 . . . 504
17.611 so_15752910_15753237_0 . . . 504
17.612 so_8813592_8816053_0 . . . 504
17.613 so_14127379_14127380_3 . . . 504
17.614 so_21766477_21767867_2 . . . 504
17.615 so_40845699_40915515_0 . . . 504
17.616 so_10833512_10833512_5 . . . 504
17.617 so_19000992_19001141_2 . . . 504
17.618 so_45793411_45794804_2 . . . 504
17.619 so_7372258_7372354_1 . . . 504
17.620 so_8813592_8816053_1 . . . 504
17.621 so_21766477_21767867_3 . . . 504
17.622 so_40845699_40915515_1 . . . 504
17.623 so_24253799_24253900_0 . . . 504
17.624 so_41371714_41416686_10 . . . 504
17.625 so_13338329_13338478_0 . . . 504
17.626 so_10833512_10833512_6 . . . 504
17.627 so_17515601_17515633_0 . . . 504
17.628 so_37463861_37464714_2 . . . 504
17.629 so_19000992_19001141_3 . . . 504
17.630 so_15325441_15325722_1 . . . 504
17.631 so_7455982_7456166_1 . . . 504
17.632 so_45793411_45794804_3 . . . 504
17.633 so_7372258_7372354_2 . . . 504
17.634 so_15752910_15753237_1 . . . 504
17.635 so_24217616_24217661_0 . . . 504
17.636 so_13132075_13132223_0 . . . 504
17.637 so_8813592_8816053_2 . . . 504
17.638 so_38859102_38859414_0 . . . 504
17.639 so_14127379_14127380_4 . . . 504
17.640 so_21766477_21767867_4 . . . 504
17.641 so_24253799_24253900_1 . . . 504

17.642 so_10833512_10833512_7 . . . 504
17.643 so_37463861_37464714_3 . . . 504
17.644 so_19000992_19001141_4 . . . 504
17.645 so_8813592_8816053_3 . . . 504
17.646 so_21766477_21767867_5 . . . 504
17.647 so_10833512_10833512_8 . . . 504
17.648 so_19000992_19001141_5 . . . 504
17.649 so_22055724_22056219_0 . . . 504
17.650 so_8813592_8816053_4 . . . 504
17.651 so_18788780_18789889_0 . . . 504
17.652 so_17309708_17309820_1 . . . 504
17.653 so_21766477_21767867_6 . . . 504
17.654 so_9006663_9006761_2 . . . 504
17.655 so_10833512_10833512_9 . . . 504
17.656 so_19000992_19001141_6 . . . 504
17.657 so_22055724_22056219_1 . . . 504
17.658 so_7455982_7456166_2 . . . 504
17.659 so_8813592_8816053_5 . . . 504
17.660 so_21766477_21767867_7 . . . 504
17.661 so_10833512_10833512_10 . . . 504
17.662 so_19000992_19001141_7 . . . 504
17.663 so_31681788_31681788_1 . . . 504
17.664 so_7455982_7456166_3 . . . 504
17.665 so_8813592_8816053_6 . . . 504
17.666 so_21766477_21767867_8 . . . 504
17.667 so_10833512_10833512_11 . . . 504
17.668 so_19000992_19001141_8 . . . 504
17.669 so_20119810_20121635_0 . . . 504
17.670 so_8813592_8816053_7 . . . 504
17.671 so_21766477_21767867_9 . . . 504
17.672 so_10833512_10833512_12 . . . 504
17.673 so_19000992_19001141_9 . . . 504
17.674 so_8813592_8816053_8 . . . 504
17.675 so_14882588_14884118_0 . . . 504
17.676 so_21539291_21561407_1 . . . 504
17.677 so_23462297_23462466_0 . . . 504
17.678 so_27670808_27671113_0 . . . 505
17.679 so_17296868_38110571_1 . . . 505
17.680 so_17296868_38110571_2 . . . 505
17.681 so_19987820_19988036_0 . . . 505
17.682 so_27609839_27610567_0 . . . 505
17.683 so_27609839_27610567_1 . . . 505
17.684 so_17958352_17958469_0 . . . 505
17.685 so_17958352_17958469_1 . . . 505
17.686 so_7542822_27759993_1 . . . 505
17.687 so_13950938_13951278_0 . . . 505
17.688 so_28746372_35578374_0 . . . 505
17.689 so_44905543_44906006_0 . . . 506
17.690 so_7930105_7930386_0 . . . 506
17.691 so_12095048_12095473_1 . . . 506
17.692 so_29866763_29866842_0 . . . 506

17.693 so_21933145_21933256_0 . . . 506
17.694 so_13680587_13686519_0 . . . 506
17.695 so_44397674_44398280_0 . . . 506
17.696 so_28205387_28205411_0 . . . 506
17.697 so_7027523_7028318_0 . . . 506
17.698 so_16479281_16479400_0 . . . 506
17.699 so_17730689_17730689_0 . . . 506
17.700 so_24130395_24130657_1 . . . 506
17.701 so_17552191_17552626_0 . . . 506
17.702 so_8522740_8522879_0 . . . 506
17.703 so_44759353_44760075_0 . . . 506
17.704 so_9168823_9168823_0 . . . 506

18 Language support library **507**
18.1 General . . . 507
18.2 Types . . . 507
18.3 so_36991202_36991202_3 . . . 507
18.4 so_6842880_6843474_5 . . . 507
18.5 so_1089176_1089204_0 . . . 507
18.6 so_37715537_37715537_1 . . . 507
18.7 so_34725939_34726289_5 . . . 508
18.8 so_40395066_40396865_0 . . . 508
18.9 so_33020532_33020942_1 . . . 508
18.10 so_24396987_24397242_0 . . . 508
18.11 so_21909837_21910159_0 . . . 508
18.12 so_46331625_46331907_2 . . . 508
18.13 so_38615436_38616085_1 . . . 508
18.14 so_9169453_9169978_3 . . . 508
18.15 so_20773783_20774847_0 . . . 508
18.16 so_23749822_23749908_1 . . . 508
18.17 so_23749822_23749989_1 . . . 508
18.18 so_23749822_23749870_0 . . . 508
18.19 so_20808905_20808905_1 . . . 508
18.20 so_32915434_32915587_0 . . . 508
18.21 so_43040231_43040436_0 . . . 508
18.22 so_27734245_27734325_0 . . . 508
18.23 so_32021860_32022085_0 . . . 508
18.24 so_15823217_15823253_0 . . . 508
18.25 so_11151687_11152454_0 . . . 508
18.26 so_8019420_8019446_0 . . . 508
18.27 so_27405418_27406408_0 . . . 508
18.28 so_33585094_33585152_0 . . . 508
18.29 so_33585094_33585265_0 . . . 508
18.30 so_33585094_33585169_0 . . . 508
18.31 so_12498709_12499245_0 . . . 508
18.32 so_23749822_23749908_2 . . . 508
18.33 so_11151687_11152454_1 . . . 508
18.34 so_27405418_27406408_1 . . . 508
18.35 so_43216468_43217371_1 . . . 508
18.36 so_28696522_28696670_2 . . . 508
18.37 so_9953068_9954287_1 . . . 508

18.38 so_5547664_5547677_4 . . . 508
18.39 so_17069315_17069469_0 . . . 508
18.40 so_14189709_14189818_0 . . . 508
18.41 Implementation properties . . . 508
18.42 so_16122912_22468790_0 . . . 508
18.43 so_23070099_23070776_0 . . . 509
18.44 so_29392607_29393566_0 . . . 509
18.45 so_31527219_31527797_4 . . . 509
18.46 so_16505782_16505879_0 . . . 509
18.47 so_2738435_2738435_1 . . . 509
18.48 so_16122912_16519653_4 . . . 509
18.49 so_16122912_22468790_1 . . . 509
18.50 so_16122912_16519653_3 . . . 509
18.51 so_29392607_29393566_2 . . . 509
18.52 so_35575276_35575276_1 . . . 509
18.53 so_9201865_9202027_0 . . . 509
18.54 so_29392607_29393566_1 . . . 510
18.55 so_35575276_35575276_2 . . . 510
18.56 so_16122912_16519653_5 . . . 510
18.57 so_35575276_35575276_3 . . . 510
18.58 so_15783525_15783662_2 . . . 510
18.59 so_41303762_41303872_0 . . . 510
18.60 so_42405800_42406650_2 . . . 510
18.61 so_15783525_15783662_7 . . . 510
18.62 so_15783525_15783662_5 . . . 511
18.63 so_42405800_42406650_3 . . . 511
18.64 so_42405800_42406650_1 . . . 511
18.65 so_15783525_15783662_6 . . . 511
18.66 so_15783525_15783662_8 . . . 511
18.67 so_19609186_19609187_0 . . . 511
18.68 so_19609186_19609187_1 . . . 511
18.69 so_27630106_27630154_0 . . . 511
18.70 so_19609186_19609187_2 . . . 511
18.71 so_34611181_34613124_4 . . . 511
18.72 so_27630106_27630154_1 . . . 511
18.73 so_19609186_19609187_3 . . . 511
18.74 so_27630106_27630154_2 . . . 511
18.75 so_23802294_23810181_5 . . . 511
18.76 so_38798791_38798823_2 . . . 511
18.77 so_42926763_42926875_5 . . . 511
18.78 so_12278039_12278086_0 . . . 511
18.79 so_3949471_3972350_2 . . . 511
18.80 so_24157094_37130892_2 . . . 511
18.81 so_17780368_17780820_2 . . . 511
18.82 so_14203654_14203676_0 . . . 511
18.83 so_14203654_14203676_1 . . . 512
18.84 so_14203654_14205392_5 . . . 512
18.85 so_14203654_14205392_1 . . . 512
18.86 so_14203654_14205392_0 . . . 512
18.87 so_23802294_23810181_2 . . . 513
18.88 so_3949471_3972350_3 . . . 513

18.89 so_3949471_3972350_1 . . . 513
18.90 so_38795544_38796216_1 . . . 513
18.91 so_38795544_38796216_2 . . . 513
18.92 so_38795544_38796216_3 . . . 513
18.93 so_23802294_23810181_3 . . . 514
18.94 so_38798791_38798823_0 . . . 514
18.95 so_42926763_42926875_6 . . . 514
18.96 so_42926763_42926875_3 . . . 514
18.97 so_24157094_37130892_0 . . . 514
18.98 so_23802294_23810181_4 . . . 514
18.99 so_38798791_38798823_3 . . . 514
18.100 so_38798791_38798823_1 . . . 514
18.101 so_42926763_42926875_4 . . . 514
18.102 so_24157094_37130892_1 . . . 514
18.103 so_24157094_37130892_3 . . . 514
18.104 so_17780368_17780820_3 . . . 514
18.105 so_17780368_17780820_1 . . . 514
18.106 so_13272959_13272959_1 . . . 514
18.107 so_13272959_13272959_2 . . . 514
18.108 so_19609186_19609187_4 . . . 515
18.109 so_38795544_38796216_4 . . . 515
18.110 so_12278039_12278086_1 . . . 515
18.111 so_26304506_26304633_8 . . . 517
18.112 so_13150449_13208789_2 . . . 517
18.113 so_34389262_34389400_7 . . . 518
18.114 Integer types . . . 518
18.115 so_30472731_30472768_0 . . . 519
18.116 so_22363102_22363263_1 . . . 519
18.117 so_22363102_22363263_0 . . . 519
18.118 so_34588650_34588805_2 . . . 519
18.119 so_41674239_41993391_0 . . . 519
18.120 so_15039077_15039190_1 . . . 519
18.121 so_34424726_34425285_0 . . . 519
18.122 so_34468090_34468484_0 . . . 519
18.123 so_5381882_30322474_1 . . . 519
18.124 so_5135734_5136535_1 . . . 519
18.125 so_31037764_31037977_0 . . . 519
18.126 so_15911714_16006093_2 . . . 519
18.127 Start and termination . . . 519
18.128 so_4769229_4769305_4 . . . 519
18.129 so_4769229_4769269_0 . . . 519
18.130 so_4769229_4769269_1 . . . 519
18.131 so_28035112_28035275_0 . . . 519
18.132 so_8804643_8804701_0 . . . 519
18.133 so_8553357_8554056_2 . . . 519
18.134 so_14030273_14030299_0 . . . 520
18.135 so_26951007_26951989_1 . . . 520
18.136 so_4582012_4582026_2 . . . 520
18.137 so_40896600_40896600_0 . . . 520
18.138 so_18317564_18317739_0 . . . 520
18.139 so_22586250_22586378_1 . . . 520

18.140 so_7255644_7255794_0 . . . 520
18.141 so_1188335_1188497_3 . . . 520
18.142 so_1188335_1188469_0 . . . 520
18.143 so_11405507_11405948_2 . . . 520
18.144 so_11405507_11405948_0 . . . 520
18.145 so_11405507_11405948_1 . . . 520
18.146 so_11405507_11405948_3 . . . 520
18.147 so_8696698_8696722_2 . . . 520
18.148 so_8696698_8697009_0 . . . 520
18.149 so_47531058_47531058_0 . . . 520
18.150 so_15911517_15911747_3 . . . 520
18.151 so_13203439_13203499_0 . . . 520
18.152 so_204476_18721336_7 . . . 520
18.153 so_26951007_26951989_0 . . . 520
18.154 so_15859665_15859685_0 . . . 520
18.155 so_4207134_8697019_1 . . . 520
18.156 so_34912588_34913061_1 . . . 520
18.157 so_43071676_43071946_0 . . . 520
18.158 so_9758495_9758495_0 . . . 521
18.159 Dynamic memory management . . . 521
18.160 so_17589601_17589601_1 . . . 522
18.161 so_11995476_11995476_0 . . . 522
18.162 so_34907922_34907922_4 . . . 522
18.163 so_16269895_16269904_1 . . . 522
18.164 so_17419639_17419953_5 . . . 522
18.165 so_377178_377208_1 . . . 522
18.166 so_26621607_26621754_0 . . . 522
18.167 so_36371825_36373803_1 . . . 522
18.168 so_16269895_16269904_4 . . . 522
18.169 so_16269895_16269904_3 . . . 522
18.170 so_16269895_16269904_6 . . . 522
18.171 so_16269895_16269904_5 . . . 522
18.172 so_24433250_24433518_0 . . . 522
18.173 so_34813448_34813599_0 . . . 522
18.174 so_4134195_4134355_3 . . . 522
18.175 so_4134195_4134355_2 . . . 522
18.176 so_4134195_4134355_1 . . . 522
18.177 so_4134195_4134355_4 . . . 522
18.178 so_17419639_17419953_1 . . . 522
18.179 so_377178_377208_5 . . . 522
18.180 so_377178_377208_6 . . . 522
18.181 so_377178_377208_4 . . . 522
18.182 so_377178_377208_3 . . . 522
18.183 so_5889691_5890000_1 . . . 522
18.184 so_19390813_19390822_1 . . . 522
18.185 so_2306587_2306587_0 . . . 522
18.186 so_26621607_26621754_1 . . . 522
18.187 so_30071603_30071603_0 . . . 522
18.188 so_6495632_6495913_4 . . . 522
18.189 so_24846456_24846529_1 . . . 522
18.190 so_33998748_33999138_6 . . . 522

18.191 so_44161611_44167468_3 . . . 522
18.192 so_24266375_24266512_0 . . . 523
18.193 so_39992588_39992802_0 . . . 523
18.194 so_28884791_28884981_0 . . . 523
18.195 so_3821261_3821334_0 . . . 523
18.196 so_44161611_44167468_0 . . . 523
18.197 so_24603142_24603142_0 . . . 524
18.198 so_33137115_33137151_3 . . . 524
18.199 so_17419639_17419953_3 . . . 524
18.200 so_44907098_44911395_0 . . . 524
18.201 so_24846456_24846529_2 . . . 524
18.202 so_6495632_6495913_6 . . . 525
18.203 so_44161611_44167468_1 . . . 525
18.204 so_26614606_26615252_1 . . . 525
18.205 so_11781724_11782277_1 . . . 525
18.206 so_40897533_40897892_1 . . . 525
18.207 so_15412570_15413320_1 . . . 525
18.208 so_13218449_13218520_0 . . . 525
18.209 so_3675059_3677129_0 . . . 525
18.210 so_4418220_4418330_0 . . . 525
18.211 so_12072182_12917192_0 . . . 525
18.212 so_17571103_17573387_3 . . . 525
18.213 so_12919474_12919474_2 . . . 525
18.214 so_12919474_12919474_1 . . . 525
18.215 so_26614606_26615252_2 . . . 526
18.216 so_745850_751173_1 . . . 526
18.217 so_15412570_15413320_3 . . . 526
18.218 so_28187732_28187927_1 . . . 526
18.219 so_11601998_47475556_2 . . . 526
18.220 so_14659752_14659931_10 . . . 526
18.221 so_9521879_9522018_1 . . . 526
18.222 so_7991490_29494419_0 . . . 526
18.223 so_16024296_16024312_0 . . . 526
18.224 so_34939724_34939831_4 . . . 527
18.225 so_20666988_20667036_1 . . . 527
18.226 so_34939724_34939831_3 . . . 527
18.227 Type identification . . . 528
18.228 so_4194191_4194266_0 . . . 528
18.229 so_32354575_32354644_0 . . . 528
18.230 so_4465872_4465965_0 . . . 528
18.231 so_25873338_25873889_2 . . . 528
18.232 so_25873338_25873889_3 . . . 528
18.233 so_25873338_25873889_4 . . . 528
18.234 so_16253207_16253207_4 . . . 529
18.235 so_3552135_3552210_0 . . . 529
18.236 so_16253207_16253207_5 . . . 529
18.237 so_3552135_3552210_1 . . . 529
18.238 so_32354575_32354724_0 . . . 529
18.239 so_15279547_15279658_4 . . . 529
18.240 so_34030309_34030360_3 . . . 529
18.241 Exception handling . . . 530

18.242 so_35039927_35039927_3 . . . 531
18.243 so_35039927_35043080_3 . . . 531
18.244 so_21644735_21649810_0 . . . 531
18.245 so_6864011_6864720_1 . . . 531
18.246 so_38479375_38482856_3 . . . 531
18.247 so_33492796_33493378_2 . . . 531
18.248 so_15449629_15449689_3 . . . 532
18.249 so_10055317_10055463_0 . . . 532
18.250 so_8553357_8554056_1 . . . 532
18.251 so_15367060_15367344_0 . . . 532
18.252 so_15798549_15801086_1 . . . 533
18.253 so_15367060_15367344_1 . . . 533
18.254 so_27741423_27741423_0 . . . 533
18.255 so_41168218_41168586_1 . . . 533
18.256 so_41168218_41168586_2 . . . 533
18.257 so_41168218_41168586_3 . . . 533
18.258 so_41168218_41168586_4 . . . 533
18.259 so_41168218_41168586_5 . . . 533
18.260 so_16915796_16916241_2 . . . 533
18.261 so_41168218_41168586_6 . . . 533
18.262 so_12690698_12691011_3 . . . 533
18.263 so_21720579_21722226_2 . . . 533
18.264 so_14775411_14775411_0 . . . 533
18.265 so_28267484_28267655_1 . . . 534
18.266 so_26217260_26217456_2 . . . 535
18.267 so_25324262_25324380_0 . . . 535
18.268 so_25324582_25325041_2 . . . 535
18.269 so_26217260_26217456_1 . . . 535
18.270 so_25324262_25324380_3 . . . 535
18.271 so_22002331_22002570_0 . . . 535
18.272 so_22002331_22002567_0 . . . 535
18.273 so_25324582_25325041_5 . . . 535
18.274 Initializer lists . . . 535
18.275 so_31724863_31725138_1 . . . 536
18.276 so_44762154_44762820_3 . . . 536
18.277 so_20501638_20501979_5 . . . 536
18.278 so_16894566_16894703_1 . . . 536
18.279 so_27157202_27157506_0 . . . 536
18.280 so_17623098_17623839_0 . . . 536
18.281 so_21104923_21104979_0 . . . 536
18.282 so_17518019_17518061_0 . . . 536
18.283 so_15180990_15181074_0 . . . 536
18.284 so_27496004_28115954_1 . . . 536
18.285 so_26021209_28244940_0 . . . 536
18.286 so_25457178_25457259_0 . . . 536
18.287 so_19066974_19067212_2 . . . 536
18.288 so_24109445_24109747_4 . . . 536
18.289 so_24109445_24109747_2 . . . 536
18.290 so_28318799_28323698_0 . . . 536
18.291 so_35696235_35697201_1 . . . 536
18.292 so_15623472_15623829_1 . . . 536

18.293 so_42896993_42897109_1 . . . 536
18.294 so_45880152_45880569_2 . . . 536
18.295 Other runtime support . . . 536
18.296 so_23373742_23375539_2 . . . 536
18.297 so_23373742_23375539_3 . . . 536
18.298 so_13223486_13223529_0 . . . 537
18.299 so_13097167_13097367_2 . . . 537
18.300 so_18491805_18493152_4 . . . 537
18.301 so_36880961_36881048_0 . . . 537
18.302 so_9121817_9121926_0 . . . 537
18.303 so_1353407_8108285_3 . . . 537
18.304 so_1353407_1353407_0 . . . 537
18.305 so_39732045_39736048_0 . . . 537
18.306 so_19876414_19876502_4 . . . 537
18.307 so_23393599_23393988_1 . . . 537
18.308 so_27802892_27805575_0 . . . 537
18.309 so_16272530_16272588_1 . . . 537
18.310 so_25723736_25724185_0 . . . 537
18.311 so_9058206_9059004_0 . . . 537
18.312 so_30292642_30292673_0 . . . 537
18.313 so_28374800_28375004_3 . . . 537
18.314 so_22753707_22755079_0 . . . 537
18.315 so_44369648_44370539_2 . . . 537
18.316 so_44369648_44370539_3 . . . 537
18.317 so_30875069_30875579_1 . . . 537
18.318 so_30875069_30875592_0 . . . 537
18.319 so_24631444_24635016_0 . . . 537
18.320 so_24631444_24635016_1 . . . 537
18.321 so_343219_9085993_0 . . . 537
18.322 so_31521182_31521467_0 . . . 537

19 Diagnostics library **539**
19.1 General . . . 539
19.2 so_47592423_47596356_0 . . . 539
19.3 Exception classes . . . 539
19.4 so_13659166_13659189_0 . . . 541
19.5 so_25984156_25984194_0 . . . 541
19.6 so_45611464_45613796_2 . . . 542
19.7 so_7624652_7624720_0 . . . 542
19.8 Assertions . . . 543
19.9 so_1571340_43877628_0 . . . 543
19.10 so_43011737_43011752_1 . . . 543
19.11 so_33908007_33908091_3 . . . 543
19.12 Error numbers . . . 543
19.13 so_14556336_14556421_0 . . . 543
19.14 System error support . . . 543
19.15 so_17458421_17474694_1 . . . 543
19.16 so_28746372_35578374_1 . . . 546
19.17 so_16052163_16052476_1 . . . 547
19.18 so_15279547_15279658_2 . . . 547
19.19 so_34030309_34030360_1 . . . 547

19.20 so_27760498_27760498_1 . . . 548
19.21 so_27760498_27760498_2 . . . 548
19.22 so_13950938_13951278_1 . . . 548
19.23 so_31433911_31438404_0 . . . 549
19.24 so_13885151_13885237_1 . . . 550
19.25 so_17458421_17474694_0 . . . 553
19.26 so_17458421_17458421_0 . . . 554

20 General utilities library **555**
20.1 General . . . 555
20.2 Utility components . . . 555
20.3 so_37668236_41351312_1 . . . 555
20.4 so_14601469_14601722_1 . . . 558
20.5 so_43096640_43096640_2 . . . 558
20.6 so_23969468_23969583_1 . . . 558
20.7 so_10335916_10336097_0 . . . 558
20.8 so_43096640_43096640_5 . . . 559
20.9 so_31577762_31578597_0 . . . 559
20.10 so_47374251_47374659_0 . . . 559
20.11 so_5529881_5529970_0 . . . 559
20.12 so_25803403_25803403_19 . . . 559
20.13 so_43819314_44692728_8 . . . 559
20.14 so_39280993_39280993_0 . . . 560
20.15 Pairs . . . 560
20.16 so_45268954_45268954_0 . . . 561
20.17 so_30056216_30056521_3 . . . 561
20.18 so_25991582_25993289_0 . . . 561
20.19 so_21405674_21407037_1 . . . 561
20.20 so_25991582_25993289_1 . . . 561
20.21 so_21405674_21407037_2 . . . 561
20.22 so_25991582_25993289_2 . . . 561
20.23 so_21405674_21407037_3 . . . 561
20.24 so_8135555_8137176_2 . . . 561
20.25 so_12214134_12215308_1 . . . 561
20.26 so_10243856_10244048_0 . . . 561
20.27 so_8137917_8137917_0 . . . 561
20.28 so_12214134_12215308_3 . . . 562
20.29 so_2695607_2695628_0 . . . 563
20.30 so_15096358_15096441_0 . . . 563
20.31 so_15096358_15096441_1 . . . 563
20.32 so_36235944_36236254_3 . . . 563
20.33 so_36235944_36236254_2 . . . 563
20.34 Tuples . . . 565
20.35 so_13902910_13903068_1 . . . 565
20.36 so_39068546_39068642_3 . . . 565
20.37 so_39068546_39068874_0 . . . 565
20.38 so_4782757_27891048_1 . . . 567
20.39 so_38392092_38392092_2 . . . 568
20.40 so_24918426_24919135_0 . . . 568
20.41 so_24918426_24919135_1 . . . 568
20.42 so_34542799_34569403_3 . . . 568

20.43 so_5481658_5483008_0 . . . 568
20.44 so_34542799_34569403_2 . . . 568
20.45 so_5481658_5483008_1 . . . 568
20.46 so_34542799_34542799_1 . . . 568
20.47 so_5481658_5483008_2 . . . 568
20.48 so_5481658_5483008_3 . . . 568
20.49 so_34542799_34569403_0 . . . 568
20.50 so_34542799_34542799_4 . . . 568
20.51 so_19111028_19111069_1 . . . 569
20.52 so_34542799_34542799_2 . . . 570
20.53 so_34542799_34542799_0 . . . 570
20.54 so_36839648_36840371_0 . . . 571
20.55 so_36839648_36840371_1 . . . 572
20.56 so_37932676_37932948_0 . . . 572
20.57 so_40584368_40584504_1 . . . 573
20.58 so_41708491_41708972_0 . . . 573
20.59 so_41708491_41708982_0 . . . 573
20.60 so_33342591_33342789_0 . . . 573
20.61 so_3313479_3313638_0 . . . 574
20.62 so_32009085_32009446_0 . . . 574
20.63 so_23827036_23827203_0 . . . 574
20.64 so_23827036_23827203_1 . . . 574
20.65 so_19111028_19111069_3 . . . 575
20.66 Compile-time integer sequences . . . 575
20.67 so_38146184_38146424_0 . . . 576
20.68 Class template `bitset` . . . 576
20.69 so_29483123_29483338_0 . . . 578
20.70 so_8165980_8166974_0 . . . 578
20.71 so_9001404_9001698_0 . . . 578
20.72 so_29483123_29483338_1 . . . 578
20.73 so_29483123_29483123_0 . . . 578
20.74 so_36988411_36988544_0 . . . 578
20.75 so_9001404_9001698_1 . . . 578
20.76 so_17738439_17738467_1 . . . 581
20.77 so_7129488_7129504_2 . . . 582
20.78 so_46367660_46371205_0 . . . 582
20.79 so_40758070_40758374_1 . . . 582
20.80 so_40758070_40758374_2 . . . 582
20.81 so_8287188_8287231_0 . . . 582
20.82 Memory . . . 583
20.83 so_15349326_15349503_1 . . . 588
20.84 so_8696452_8696569_1 . . . 588
20.85 so_43243129_43243562_0 . . . 588
20.86 so_35081546_35082217_3 . . . 589
20.87 so_26472434_26473169_2 . . . 592
20.88 so_33265165_33266918_0 . . . 592
20.89 so_33265165_33265165_1 . . . 592
20.90 so_30036904_30038341_0 . . . 592
20.91 so_25903968_25904184_6 . . . 592
20.92 so_46409188_46409188_9 . . . 592
20.93 so_9521879_9522152_1 . . . 594

20.94 so_13785690_13785690_0 . . . 594
20.95 so_13785690_13785737_0 . . . 594
20.96 so_36540378_36540825_1 . . . 594
20.97 so_25680528_25680660_0 . . . 594
20.98 so_46399544_46400243_1 . . . 594
20.99 so_46399544_46400243_2 . . . 594
20.100 so_27217656_27217701_0 . . . 594
20.101 so_35832358_35832454_0 . . . 594
20.102 so_13097268_13100870_1 . . . 594
20.103 so_38650310_38701636_4 . . . 594
20.104 so_42578832_42579091_1 . . . 594
20.105 so_32961922_32961922_0 . . . 594
20.106 so_47072261_47104045_0 . . . 594
20.107 so_9521879_9522018_0 . . . 594
20.108 so_25680528_25680660_1 . . . 594
20.109 so_14118580_14118791_1 . . . 594
20.110 so_46823224_46823285_0 . . . 594
20.111 so_3104543_3104591_0 . . . 594
20.112 so_42578832_42579091_2 . . . 594
20.113 so_13137766_13138095_1 . . . 594
20.114 so_30354113_30354909_0 . . . 595
20.115 so_3264299_3264299_0 . . . 596
20.116 so_11985330_11985331_0 . . . 596
20.117 so_23749822_23749908_0 . . . 597
20.118 so_23749822_23749989_0 . . . 597
20.119 so_5813700_5813736_1 . . . 597
20.120 so_22406278_22406540_0 . . . 597
20.121 so_22406278_22406540_1 . . . 597
20.122 so_5644937_5644963_0 . . . 597
20.123 so_22406278_22406540_2 . . . 597
20.124 so_15157591_15157591_0 . . . 597
20.125 so_18347914_18348009_0 . . . 597
20.126 so_39595103_39595592_0 . . . 598
20.127 Smart pointers . . . 598
20.128 so_32738518_32739887_1 . . . 598
20.129 so_31749700_31750080_0 . . . 598
20.130 so_23849041_23849282_0 . . . 598
20.131 so_24061767_24061911_0 . . . 598
20.132 so_24061767_24061876_0 . . . 598
20.133 so_29922666_42990870_0 . . . 598
20.134 so_24130395_24130657_3 . . . 598
20.135 so_38027402_38027496_0 . . . 598
20.136 so_36071220_36071266_0 . . . 598
20.137 so_31749700_31750080_2 . . . 598
20.138 so_31410352_31410483_0 . . . 598
20.139 so_25803403_25803403_5 . . . 598
20.140 so_12764952_12764952_2 . . . 598
20.141 so_19747137_19747382_0 . . . 598
20.142 so_12764952_12764952_4 . . . 600
20.143 so_11455989_11456151_3 . . . 600
20.144 so_12764952_12764952_5 . . . 600

20.145 so_42226148_42226605_0 . . . 601
20.146 so_23438041_23438554_2 . . . 601
20.147 so_36850522_36851037_1 . . . 601
20.148 so_6265288_6272139_0 . . . 601
20.149 so_14063717_14067836_0 . . . 601
20.150 so_29895256_29895468_0 . . . 601
20.151 so_3899790_3900082_1 . . . 602
20.152 so_19846444_19846444_1 . . . 602
20.153 so_37679008_37679220_0 . . . 602
20.154 so_17906325_17906410_1 . . . 602
20.155 so_23613104_23625482_0 . . . 602
20.156 so_36071220_36071498_0 . . . 603
20.157 so_36401999_36404838_0 . . . 604
20.158 so_38447426_38447460_0 . . . 604
20.159 so_23438041_23438554_0 . . . 604
20.160 so_19528691_19528917_1 . . . 604
20.161 so_37788282_37788488_0 . . . 604
20.162 so_13116534_13116656_1 . . . 604
20.163 so_32735296_32735296_0 . . . 604
20.164 so_11164354_11164508_0 . . . 604
20.165 so_11164354_11164463_0 . . . 604
20.166 so_35317344_35317344_0 . . . 604
20.167 so_27302548_27302643_1 . . . 604
20.168 so_42133510_42133592_0 . . . 604
20.169 so_35317344_35317344_1 . . . 604
20.170 so_44759353_44760075_2 . . . 604
20.171 so_23438041_23438554_1 . . . 605
20.172 so_40174207_40175849_0 . . . 605
20.173 so_11651903_11652406_0 . . . 605
20.174 so_27302548_27302643_0 . . . 605
20.175 so_38447426_38447460_1 . . . 605
20.176 so_21606073_21606198_0 . . . 605
20.177 so_42133510_42133592_1 . . . 605
20.178 so_13116534_13116656_2 . . . 607
20.179 so_33229890_33230203_0 . . . 607
20.180 so_20313322_20313322_1 . . . 607
20.181 so_39290088_39290209_0 . . . 607
20.182 so_16596950_16596950_3 . . . 607
20.183 so_20313322_20313322_3 . . . 607
20.184 so_16596950_16596950_4 . . . 607
20.185 so_20313322_20313322_4 . . . 607
20.186 so_16596950_16596950_5 . . . 607
20.187 so_20313322_20313322_6 . . . 607
20.188 so_16596950_16596950_6 . . . 607
20.189 so_20313322_20313322_7 . . . 607
20.190 so_36644263_36644696_0 . . . 612
20.191 so_31345193_31347287_4 . . . 612
20.192 so_4497504_4497540_0 . . . 612
20.193 so_28113769_28113769_0 . . . 612
20.194 so_28113769_28114649_0 . . . 612
20.195 so_28113769_28114649_1 . . . 612

20.196 so_28113769_28114915_0 . . . 612
20.197 so_27730158_27735051_0 . . . 612
20.198 so_28118904_28118904_1 . . . 612
20.199 so_22927045_22927230_0 . . . 612
20.200 so_5676835_5677188_0 . . . 612
20.201 so_11491812_11491905_0 . . . 612
20.202 so_3899790_3900082_0 . . . 612
20.203 so_19846444_19846444_0 . . . 612
20.204 so_11922262_11922297_0 . . . 612
20.205 so_14109542_14109569_0 . . . 612
20.206 so_19846444_19846444_3 . . . 612
20.207 so_25655145_25655195_4 . . . 612
20.208 so_11922262_11922297_1 . . . 612
20.209 so_25655145_25655195_1 . . . 612
20.210 so_19102034_19102344_3 . . . 612
20.211 so_11164354_11164508_4 . . . 612
20.212 so_25655145_25655195_2 . . . 612
20.213 so_30798278_30798799_0 . . . 612
20.214 so_30798278_30798799_1 . . . 613
20.215 so_1403465_1403539_8 . . . 613
20.216 so_1403465_1403539_14 . . . 613
20.217 so_19102034_19102344_0 . . . 613
20.218 so_1403465_1403539_11 . . . 613
20.219 so_1403465_1403539_17 . . . 613
20.220 so_1403465_1403539_18 . . . 613
20.221 so_1403465_1403539_12 . . . 613
20.222 so_24913269_24913605_3 . . . 613
20.223 so_16794332_16794471_0 . . . 613
20.224 so_41140170_41149207_1 . . . 613
20.225 so_6322364_6324146_0 . . . 613
20.226 so_24913269_24913605_4 . . . 613
20.227 so_24913269_24913605_5 . . . 613
20.228 so_22975453_22975572_0 . . . 613
20.229 so_17148335_17148392_3 . . . 613
20.230 so_3899790_3900082_3 . . . 614
20.231 so_33080528_33080560_5 . . . 614
20.232 so_33080528_33080560_7 . . . 614
20.233 so_33080528_33080560_6 . . . 614
20.234 so_46012436_46012658_1 . . . 614
20.235 so_41851520_41852166_3 . . . 614
20.236 so_41851520_41851940_5 . . . 614
20.237 so_46647004_46647004_1 . . . 614
20.238 so_24913269_24913605_2 . . . 614
20.239 so_8287010_8287110_3 . . . 614
20.240 so_8287010_8287110_4 . . . 614
20.241 so_11651903_11652538_0 . . . 614
20.242 so_41851520_41851940_1 . . . 615
20.243 so_12402116_12402162_0 . . . 615
20.244 so_31847540_31847672_0 . . . 615
20.245 so_36370180_36370180_0 . . . 615
20.246 so_46647004_46647004_3 . . . 615

20.247 so_28118904_28119009_1 . . . 615
20.248 so_8147027_8147213_0 . . . 616
20.249 so_9302296_9303504_1 . . . 616
20.250 so_25804018_25804086_0 . . . 616
20.251 so_17104209_17104608_1 . . . 616
20.252 so_17104209_17104573_0 . . . 616
20.253 so_9675225_9675225_0 . . . 616
20.254 so_37734566_37734566_1 . . . 616
20.255 so_32783394_32783472_0 . . . 616
20.256 so_32783394_32783600_0 . . . 616
20.257 so_25804018_25804086_1 . . . 616
20.258 so_25804018_25804088_0 . . . 616
20.259 so_14080408_14080459_0 . . . 616
20.260 so_24779929_24780012_3 . . . 616
20.261 so_9200664_35204206_4 . . . 616
20.262 so_24779929_24780012_4 . . . 616
20.263 so_11079770_11080241_0 . . . 617
20.264 so_38180899_38183507_0 . . . 618
20.265 so_41851520_41851940_3 . . . 620
20.266 so_20705304_30085310_0 . . . 620
20.267 so_4173254_47514417_1 . . . 621
20.268 so_30798278_30798799_3 . . . 622
20.269 so_26123968_26124465_0 . . . 622
20.270 so_47257218_47260043_1 . . . 622
20.271 so_34213087_34213285_3 . . . 622
20.272 so_30798278_30798799_4 . . . 622
20.273 so_24851110_24869078_0 . . . 622
20.274 so_46647004_46647212_1 . . . 622
20.275 so_29182930_29184815_1 . . . 622
20.276 so_28338978_28351087_0 . . . 622
20.277 so_10338606_32201462_0 . . . 622
20.278 so_10338606_10338606_0 . . . 622
20.279 so_24420684_24420797_1 . . . 622
20.280 so_14482830_29045211_1 . . . 623
20.281 so_5676835_5677188_1 . . . 623
20.282 so_16422486_16422620_6 . . . 624
20.283 Function objects . . . 625
20.284 so_12203347_12206063_0 . . . 625
20.285 so_25299888_25299996_2 . . . 625
20.286 so_25299888_25299888_0 . . . 625
20.287 so_25299888_25299996_0 . . . 625
20.288 so_17644816_17645573_1 . . . 625
20.289 so_19720064_19851499_2 . . . 625
20.290 so_24867294_24868125_3 . . . 625
20.291 so_31809388_31809505_0 . . . 625
20.292 so_10730682_10732001_1 . . . 629
20.293 so_24867294_24868125_2 . . . 629
20.294 so_10730682_10732001_2 . . . 629
20.295 so_10730682_10732001_3 . . . 629
20.296 so_39186871_39186871_1 . . . 629
20.297 so_15310989_15310989_2 . . . 629

20.298 so_15310989_15310989_4 . . . 629
20.299 so_15310989_15310989_1 . . . 629
20.300 so_15310989_15310989_3 . . . 629
20.301 so_15310989_15310989_5 . . . 629
20.302 so_39146874_39147737_4 . . . 629
20.303 so_14769437_14769630_0 . . . 629
20.304 so_15080200_15080536_2 . . . 629
20.305 so_15080200_15080536_1 . . . 629
20.306 so_15080200_15080536_3 . . . 629
20.307 so_15080200_15080536_5 . . . 629
20.308 so_15080200_15080536_4 . . . 629
20.309 so_7408263_7409500_4 . . . 629
20.310 so_7408263_7409500_3 . . . 629
20.311 so_9339829_9341064_2 . . . 629
20.312 so_37940674_37963648_2 . . . 629
20.313 so_37940674_37963648_3 . . . 629
20.314 so_15080015_15080378_7 . . . 629
20.315 so_15080015_15080378_6 . . . 629
20.316 so_15080015_15080378_5 . . . 629
20.317 so_15080015_15080378_4 . . . 629
20.318 so_15080015_15080378_3 . . . 629
20.319 so_11795349_11795528_0 . . . 629
20.320 so_4821268_4821663_0 . . . 629
20.321 so_12638393_12638393_2 . . . 629
20.322 so_12638393_12638393_1 . . . 629
20.323 so_12638393_12638393_3 . . . 629
20.324 so_12638393_12638393_4 . . . 629
20.325 so_12638393_12638393_5 . . . 629
20.326 so_28158861_28159027_6 . . . 629
20.327 so_28158861_28159027_2 . . . 629
20.328 so_28158861_28159027_3 . . . 629
20.329 so_28158861_28159027_5 . . . 629
20.330 so_28158861_28159027_4 . . . 629
20.331 so_26692378_26692478_3 . . . 629
20.332 so_26692378_26692478_2 . . . 629
20.333 so_35439455_35439967_2 . . . 629
20.334 so_15264003_15264126_4 . . . 629
20.335 so_15264003_15264126_1 . . . 629
20.336 so_15264003_15264126_5 . . . 629
20.337 so_15264003_15264126_2 . . . 629
20.338 so_15264003_15264126_3 . . . 629
20.339 so_31189812_31189812_1 . . . 630
20.340 so_15122410_15122462_1 . . . 634
20.341 so_1380567_1380586_1 . . . 634
20.342 so_4607418_4608296_0 . . . 635
20.343 so_1098966_1099080_0 . . . 635
20.344 so_3455706_3455812_0 . . . 635
20.345 so_30617158_30617677_1 . . . 635
20.346 so_5322227_5322329_0 . . . 635
20.347 so_44907303_44907759_3 . . . 635
20.348 so_28811034_28811449_17 . . . 635

20.349 so_28811034_28811449_18 . . . 635
20.350 so_12031781_12032000_0 . . . 635
20.351 so_13380063_13380063_0 . . . 635
20.352 so_17644816_17645573_2 . . . 635
20.353 so_16301088_16301421_1 . . . 635
20.354 so_18597429_18597508_1 . . . 635
20.355 so_18597429_18597508_0 . . . 635
20.356 so_22824974_22825016_1 . . . 635
20.357 so_39545246_39545271_1 . . . 635
20.358 so_21192659_21193316_0 . . . 639
20.359 so_12760977_12761009_0 . . . 639
20.360 so_40965191_40965252_0 . . . 639
20.361 so_5065171_5065363_0 . . . 639
20.362 so_5065171_5065363_1 . . . 639
20.363 so_19409904_19409992_1 . . . 639
20.364 so_26298377_26652633_1 . . . 639
20.365 so_40670758_40670911_0 . . . 639
20.366 so_15310989_15311159_0 . . . 640
20.367 so_27962421_27963240_1 . . . 640
20.368 so_7408263_7409500_1 . . . 640
20.369 so_23588574_23588574_1 . . . 640
20.370 so_16164769_16165105_0 . . . 640
20.371 so_28382587_28382673_0 . . . 640
20.372 so_23588574_23588574_0 . . . 640
20.373 so_8746260_8746431_0 . . . 640
20.374 so_21059115_21059616_0 . . . 640
20.375 so_10053998_10054118_0 . . . 640
20.376 so_45698527_45699358_0 . . . 640
20.377 so_23588574_23588653_0 . . . 640
20.378 so_5126219_5126331_0 . . . 640
20.379 so_16164769_16165105_1 . . . 640
20.380 so_37940674_37963648_0 . . . 641
20.381 so_28528210_28528400_0 . . . 641
20.382 so_12968135_13041305_0 . . . 641
20.383 so_12638393_12638715_1 . . . 641
20.384 so_28528210_28528400_1 . . . 641
20.385 so_28528210_28528400_2 . . . 641
20.386 so_7526853_7527115_1 . . . 641
20.387 so_33822271_33822304_1 . . . 641
20.388 so_10170087_10170224_0 . . . 643
20.389 so_38302471_38303037_1 . . . 643
20.390 so_25395605_25395605_3 . . . 643
20.391 so_39146874_39147737_1 . . . 643
20.392 so_20129601_29457380_3 . . . 643
20.393 so_10730682_10732001_4 . . . 643
20.394 so_10730682_10731800_1 . . . 643
20.395 so_9339829_9341064_1 . . . 643
20.396 so_25385876_25386885_1 . . . 643
20.397 so_32871606_32872389_1 . . . 643
20.398 so_38760784_38763375_4 . . . 643
20.399 so_23284971_23286933_1 . . . 643

20.400 so_28167166_28167455_4 . . . 643
20.401 so_10730682_10732001_0 . . . 643
20.402 so_23613574_23613574_25 . . . 643
20.403 so_23613574_23613574_21 . . . 643
20.404 so_23613574_23613574_10 . . . 643
20.405 so_23613574_23613574_40 . . . 643
20.406 so_23613574_23613574_5 . . . 643
20.407 so_23613574_23613574_55 . . . 643
20.408 so_11126238_13000025_0 . . . 643
20.409 so_36617358_36619948_0 . . . 643
20.410 so_36617358_36617585_2 . . . 643
20.411 so_13680587_13680696_0 . . . 644
20.412 so_13241806_13241886_0 . . . 644
20.413 so_25421346_25421519_1 . . . 644
20.414 so_28011405_28011588_1 . . . 644
20.415 so_16823501_16823501_0 . . . 644
20.416 so_38302471_38303037_0 . . . 644
20.417 so_31172921_31172921_0 . . . 644
20.418 so_31172921_31173681_2 . . . 644
20.419 so_20129601_29457380_1 . . . 644
20.420 so_10730682_10731800_0 . . . 644
20.421 so_9339829_9343400_0 . . . 644
20.422 so_9339829_9341064_0 . . . 644
20.423 so_25385876_25386885_0 . . . 644
20.424 so_32871606_32872389_0 . . . 644
20.425 so_22146749_22147471_6 . . . 644
20.426 so_30393285_30394755_0 . . . 644
20.427 so_12145035_12145992_0 . . . 644
20.428 so_23284971_23286933_0 . . . 644
20.429 so_25879865_25880625_2 . . . 644
20.430 so_16823501_16823501_2 . . . 644
20.431 so_22146749_22147471_2 . . . 644
20.432 so_33198974_33201576_1 . . . 644
20.433 so_27953336_27953906_0 . . . 644
20.434 so_22146749_22147471_3 . . . 644
20.435 so_13885151_13885237_2 . . . 645
20.436 so_22998364_23002021_0 . . . 645
20.437 so_23357236_23365730_2 . . . 645
20.438 so_23357236_23365730_3 . . . 645
20.439 so_26692378_26692478_0 . . . 645
20.440 so_33069029_33069029_0 . . . 645
20.441 so_13680587_13686519_2 . . . 645
20.442 so_13680587_13686519_3 . . . 645
20.443 so_5812483_5812749_0 . . . 645
20.444 so_46452012_46452188_0 . . . 646
20.445 so_10225760_10225760_2 . . . 646
20.446 so_32380081_32380145_0 . . . 646
20.447 so_14669995_14670387_0 . . . 646
20.448 so_11990794_11990952_0 . . . 646
20.449 Metaprogramming and type traits . . . 647
20.450 so_18072661_18072866_2 . . . 647

20.451 so_34612344_34612550_1 . . . 647
20.452 so_40584368_40584504_2 . . . 647
20.453 so_25345486_25345486_0 . . . 651
20.454 so_36523038_36523039_0 . . . 651
20.455 so_25345206_25345359_0 . . . 651
20.456 so_17065704_17066793_3 . . . 651
20.457 so_41029092_41030249_0 . . . 651
20.458 so_34612344_34612550_0 . . . 651
20.459 so_30020005_30020407_0 . . . 651
20.460 so_25345486_25345871_0 . . . 651
20.461 so_30020005_30020407_1 . . . 651
20.462 so_42441791_42441791_0 . . . 651
20.463 so_1831991_1832104_0 . . . 651
20.464 so_35947296_35947513_17 . . . 651
20.465 so_16619048_16621661_0 . . . 651
20.466 so_33146096_33148575_0 . . . 651
20.467 so_21080532_21080853_1 . . . 651
20.468 so_45071876_45072047_0 . . . 653
20.469 so_27244588_27245211_0 . . . 653
20.470 so_12754886_12755118_1 . . . 653
20.471 so_12754886_12755118_3 . . . 653
20.472 so_23716391_23716762_1 . . . 653
20.473 so_44863382_44863678_1 . . . 653
20.474 so_33480858_33480951_0 . . . 653
20.475 so_33480858_33480961_1 . . . 653
20.476 so_2328408_2328461_0 . . . 653
20.477 so_40382838_40383196_0 . . . 653
20.478 so_15064535_15066266_3 . . . 653
20.479 so_26648817_26649961_2 . . . 653
20.480 so_36356115_36356212_0 . . . 653
20.481 so_35496609_35497520_0 . . . 653
20.482 so_11919903_11919903_0 . . . 653
20.483 so_4178175_7189821_2 . . . 653
20.484 so_9782625_9782625_0 . . . 653
20.485 so_42687094_42687094_0 . . . 653
20.486 so_42048621_42084008_1 . . . 653
20.487 so_13955677_13955874_0 . . . 653
20.488 so_35708014_35708482_0 . . . 653
20.489 so_40821950_40822131_1 . . . 653
20.490 so_7360752_11814437_3 . . . 653
20.491 so_7360752_11814437_1 . . . 653
20.492 so_9747406_9747527_4 . . . 653
20.493 so_8955052_8955207_0 . . . 653
20.494 so_5430801_5453372_0 . . . 653
20.495 so_16232547_16233624_0 . . . 653
20.496 so_47034088_47236038_3 . . . 653
20.497 so_36098055_36098055_1 . . . 653
20.498 so_32694114_32694707_2 . . . 653
20.499 so_40959503_40959503_0 . . . 653
20.500 so_27339014_27339171_0 . . . 653
20.501 so_22620517_22620776_2 . . . 653

20.502 so_11091385_11091445_0 . . . 653
20.503 so_36354041_36354041_0 . . . 653
20.504 so_14009120_14009579_0 . . . 653
20.505 so_25803403_25803403_16 . . . 653
20.506 so_11979153_11979192_0 . . . 653
20.507 so_19611478_19611775_1 . . . 653
20.508 so_34612344_34612452_0 . . . 653
20.509 so_29759441_29759556_1 . . . 653
20.510 so_29759441_29759556_0 . . . 653
20.511 so_19775954_19775988_0 . . . 653
20.512 so_25127204_25127444_2 . . . 653
20.513 so_39763548_39764671_1 . . . 653
20.514 so_36962583_36962583_0 . . . 653
20.515 so_43213997_43219391_0 . . . 653
20.516 so_4275861_4417899_4 . . . 653
20.517 so_25323026_25323153_0 . . . 653
20.518 so_33909390_33917875_3 . . . 653
20.519 so_36529535_36529535_0 . . . 653
20.520 so_34939724_34939831_0 . . . 653
20.521 so_36016770_36016914_0 . . . 653
20.522 so_27556840_27557375_1 . . . 653
20.523 so_14974591_14974859_0 . . . 653
20.524 so_9734175_9734373_0 . . . 653
20.525 so_25902512_25902703_3 . . . 653
20.526 so_25902512_25902703_1 . . . 653
20.527 so_23462067_23640320_1 . . . 653
20.528 so_45653336_45655346_1 . . . 653
20.529 so_38103578_38104223_5 . . . 653
20.530 so_7536153_7536230_3 . . . 653
20.531 so_42034848_42035156_1 . . . 653
20.532 so_42034848_42035156_0 . . . 653
20.533 so_16542291_16542385_0 . . . 653
20.534 so_28900153_28905103_1 . . . 653
20.535 so_31398388_31398779_0 . . . 653
20.536 so_20816117_20816449_1 . . . 659
20.537 so_20816117_20816449_0 . . . 659
20.538 so_23003942_23013978_0 . . . 659
20.539 so_21405674_21407037_4 . . . 659
20.540 so_21405674_21407037_5 . . . 659
20.541 so_21405674_21407037_7 . . . 659
20.542 so_21405674_21407037_6 . . . 659
20.543 so_25803403_25803813_1 . . . 659
20.544 so_25803403_25803403_20 . . . 659
20.545 so_25803403_25803403_18 . . . 659
20.546 so_14815998_14815999_1 . . . 659
20.547 so_14815998_14815999_2 . . . 659
20.548 so_14815998_14815999_0 . . . 659
20.549 so_14815998_14815999_3 . . . 659
20.550 so_28085847_28087336_0 . . . 659
20.551 so_28085847_28086046_0 . . . 659
20.552 so_28085847_28086046_1 . . . 659

20.553 so_28085847_28087336_2 . . . 659
20.554 so_28085847_28087336_1 . . . 659
20.555 so_24918426_24919135_2 . . . 659
20.556 so_33058915_33059200_0 . . . 659
20.557 so_33058915_33058915_0 . . . 659
20.558 so_33058915_33058915_1 . . . 659
20.559 so_20815921_20815977_3 . . . 659
20.560 so_20815921_20815977_2 . . . 659
20.561 so_8298214_8298487_0 . . . 659
20.562 so_27949748_27949748_0 . . . 660
20.563 so_15090216_15090354_0 . . . 660
20.564 so_13816778_13817530_0 . . . 660
20.565 so_23003942_23013978_1 . . . 661
20.566 so_21405674_21407037_9 . . . 661
20.567 so_21405674_21407037_8 . . . 661
20.568 so_20860535_20860535_0 . . . 661
20.569 so_20860535_20860535_1 . . . 661
20.570 so_24918426_24919135_3 . . . 661
20.571 so_10965325_10965371_0 . . . 665
20.572 so_24045850_24046184_2 . . . 665
20.573 so_46457386_46457745_0 . . . 666
20.574 so_14005020_14005158_0 . . . 666
20.575 Compile-time rational arithmetic . . . 666
20.576 so_18513107_18513968_0 . . . 666
20.577 so_22917060_22917247_0 . . . 668
20.578 Time utilities . . . 669
20.579 so_40930541_40931104_0 . . . 672
20.580 so_13263277_13263328_4 . . . 672
20.581 so_13263277_13263328_3 . . . 672
20.582 so_46740302_46740824_0 . . . 672
20.583 so_12411370_12411370_0 . . . 673
20.584 so_16422703_16422780_3 . . . 673
20.585 so_16422703_16422780_1 . . . 673
20.586 so_14777040_14788089_1 . . . 675
20.587 so_42746099_42746099_0 . . . 675
20.588 so_14808663_14809603_2 . . . 676
20.589 so_21626236_21626576_0 . . . 676
20.590 so_30555121_30555257_0 . . . 679
20.591 so_30555121_30555275_0 . . . 679
20.592 so_13263277_13263328_0 . . . 683
20.593 so_46740302_46741303_1 . . . 683
20.594 so_38252022_38253266_2 . . . 683
20.595 so_30882953_33758235_3 . . . 683
20.596 so_13263277_13263328_1 . . . 684
20.597 so_46740302_46741303_3 . . . 684
20.598 so_15817235_15817235_0 . . . 684
20.599 so_2940748_2940748_1 . . . 684
20.600 so_30882953_33758235_1 . . . 684
20.601 so_34133658_34133780_1 . . . 684
20.602 so_13263277_13263328_2 . . . 684
20.603 so_38252022_38253266_1 . . . 684

20.604 so_34823811_34824190_1 . . . 684
20.605 so_35682325_35682325_0 . . . 684
20.606 Class template `scoped_allocator_adaptor` . . . 685
20.607 so_15729765_15730055_0 . . . 685
20.608 so_12556638_12557286_2 . . . 685
20.609 so_25806959_25806959_4 . . . 687
20.610 Class `type_index` . . . 691
20.611 so_16253207_16253207_3 . . . 692

21 Strings library **693**
21.1 General . . . 693
21.2 so_12290276_12290372_0 . . . 693
21.3 so_9678417_9679503_2 . . . 693
21.4 so_15273964_15276769_0 . . . 693
21.5 so_9241935_9242752_0 . . . 693
21.6 so_7743356_16796851_0 . . . 693
21.7 Character traits . . . 693
21.8 so_38044834_38044974_1 . . . 693
21.9 so_27620277_27620766_1 . . . 693
21.10 so_27620277_27620766_2 . . . 693
21.11 so_27620277_27620766_0 . . . 693
21.12 so_20593878_20593934_3 . . . 693
21.13 so_10066328_10066372_0 . . . 693
21.14 so_37202716_37202863_0 . . . 693
21.15 so_43769773_43771344_1 . . . 693
21.16 so_34612344_34612453_1 . . . 693
21.17 so_19897484_19897554_1 . . . 693
21.18 so_24482728_24482877_1 . . . 695
21.19 so_21391875_21469719_0 . . . 695
21.20 so_37970997_37971922_3 . . . 696
21.21 so_30935840_30936261_0 . . . 696
21.22 so_17826821_17827365_1 . . . 696
21.23 so_4516078_4516153_0 . . . 697
21.24 so_23199107_23199281_0 . . . 697
21.25 so_43769773_43771344_0 . . . 697
21.26 String classes . . . 699
21.27 so_30685838_30686035_0 . . . 699
21.28 Class template `basic_string` . . . 703
21.29 so_9678417_9679503_1 . . . 703
21.30 so_33124479_33124747_1 . . . 703
21.31 so_33657050_33675281_9 . . . 703
21.32 so_21871902_21872347_0 . . . 707
21.33 so_39200665_39200666_0 . . . 707
21.34 so_30977974_30978292_0 . . . 707
21.35 so_34163580_34164001_0 . . . 707
21.36 so_1986966_28024825_0 . . . 707
21.37 so_30187915_30188026_0 . . . 707
21.38 so_9678417_9679503_4 . . . 707
21.39 so_33893537_34232265_0 . . . 707
21.40 so_15273964_15276769_1 . . . 707
21.41 so_33657050_33657050_1 . . . 707

21.42 so_24761896_24763545_1 . . . 707
21.43 so_2256160_2256218_0 . . . 707
21.44 so_33124479_33124544_0 . . . 707
21.45 so_7518732_7518875_0 . . . 707
21.46 so_39069377_39069417_0 . . . 707
21.47 so_14290795_14290795_0 . . . 707
21.48 so_4557009_4557156_0 . . . 707
21.49 so_30079939_30080323_1 . . . 707
21.50 so_25068637_25068713_3 . . . 707
21.51 so_12199710_12199969_0 . . . 707
21.52 so_12199710_38651410_5 . . . 707
21.53 so_12199710_12199969_1 . . . 707
21.54 so_39023367_39031767_0 . . . 707
21.55 so_39023367_39031767_3 . . . 707
21.56 so_20685327_20685704_0 . . . 707
21.57 so_20685327_20685569_1 . . . 707
21.58 so_20685327_20685569_0 . . . 707
21.59 so_20685327_20685569_2 . . . 707
21.60 so_23764785_23764853_0 . . . 707
21.61 so_10362672_10362839_0 . . . 707
21.62 so_10362672_10362839_2 . . . 707
21.63 so_10362672_10362839_3 . . . 707
21.64 so_10362672_10362839_1 . . . 707
21.65 so_3823058_3823248_0 . . . 707
21.66 so_22147925_22147925_4 . . . 707
21.67 so_22147925_22147925_2 . . . 707
21.68 so_22147925_22147925_1 . . . 707
21.69 so_8190950_8191356_2 . . . 707
21.70 so_8190950_8191356_1 . . . 707
21.71 so_28748358_28748611_0 . . . 707
21.72 so_28748358_28752639_0 . . . 707
21.73 so_28748358_28752639_4 . . . 707
21.74 so_28748358_28752639_1 . . . 707
21.75 so_3160882_3160882_0 . . . 707
21.76 so_15273964_15276769_2 . . . 707
21.77 so_24761896_24763545_2 . . . 707
21.78 so_24761896_24763545_3 . . . 707
21.79 so_24761896_24763545_4 . . . 707
21.80 so_14716053_14716174_0 . . . 707
21.81 so_44301382_44303239_0 . . . 707
21.82 so_45934295_45934606_1 . . . 707
21.83 so_45934295_45934606_2 . . . 707
21.84 so_45934295_45934606_0 . . . 707
21.85 so_37495418_37495545_0 . . . 707
21.86 so_26866438_26866812_0 . . . 708
21.87 so_12199710_12199893_2 . . . 708
21.88 so_17738439_17738729_1 . . . 708
21.89 so_17738439_17738729_0 . . . 708
21.90 so_17738439_17738467_2 . . . 708
21.91 so_28953540_28953566_1 . . . 708
21.92 so_46673520_46673624_1 . . . 708

21.93 so_12199710_12199893_3 . . . 708
21.94 so_37112686_37113417_2 . . . 708
21.95 so_38784018_38785417_0 . . . 708
21.96 so_8467049_8467073_0 . . . 708
21.97 so_45190999_45192353_0 . . . 708
21.98 so_32976919_32977187_0 . . . 708
21.99 so_9057119_9057119_1 . . . 708
21.100 so_15680489_15680881_4 . . . 708
21.101 so_18370694_18371156_14 . . . 708
21.102 so_18370694_18371156_12 . . . 708
21.103 so_19778226_19778403_0 . . . 708
21.104 so_18343518_18343629_1 . . . 708
21.105 so_39310347_39310427_2 . . . 708
21.106 so_15506275_15506302_0 . . . 708
21.107 so_34901903_34902409_1 . . . 709
21.108 so_37074905_37075372_0 . . . 709
21.109 so_12285060_12285088_0 . . . 709
21.110 so_31619467_31619921_1 . . . 709
21.111 so_12285060_12285088_4 . . . 709
21.112 so_19897484_19897554_0 . . . 709
21.113 so_31068646_31068720_0 . . . 709
21.114 so_26866438_26866812_1 . . . 709
21.115 so_19778226_19778403_1 . . . 709
21.116 so_15142603_15142704_0 . . . 710
21.117 so_29183174_29183174_1 . . . 710
21.118 so_32346143_32346399_1 . . . 710
21.119 so_13597170_13599511_2 . . . 710
21.120 so_18370694_18371156_10 . . . 710
21.121 so_18370694_18371156_2 . . . 711
21.122 so_33657050_33675281_6 . . . 711
21.123 so_483337_483517_1 . . . 712
21.124 so_31628940_31629020_1 . . . 712
21.125 so_12296896_12296923_0 . . . 712
21.126 so_31628940_31629020_2 . . . 712
21.127 so_7743356_16796851_2 . . . 712
21.128 so_13137766_13138095_0 . . . 712
21.129 so_25054302_25055430_8 . . . 712
21.130 so_25054302_25055430_4 . . . 712
21.131 so_25054302_25055430_5 . . . 712
21.132 so_25054302_25055430_10 . . . 712
21.133 so_25054302_25055430_9 . . . 712
21.134 so_25054302_25055430_3 . . . 712
21.135 so_25054302_25055430_11 . . . 712
21.136 so_11393566_11393777_0 . . . 712
21.137 so_11393566_11393777_1 . . . 712
21.138 so_11393566_11393777_2 . . . 712
21.139 so_17529243_17529403_0 . . . 713
21.140 so_17529243_17529403_1 . . . 713
21.141 so_17529243_17529349_0 . . . 713
21.142 so_39200665_39200666_12 . . . 713
21.143 so_30079644_30080040_0 . . . 713

21.144 so_25687891_25688227_5 . . . 713
21.145 so_34163580_34164001_1 . . . 713
21.146 so_18299631_18299710_8 . . . 713
21.147 so_11809849_11809849_0 . . . 713
21.148 so_12740403_41310431_0 . . . 713
21.149 so_12740403_12740932_0 . . . 713
21.150 so_1986966_1986974_0 . . . 713
21.151 so_13521961_13522000_0 . . . 713
21.152 so_27551900_27552205_2 . . . 713
21.153 so_33083245_33083361_0 . . . 713
21.154 so_7554039_7554172_1 . . . 713
21.155 so_34313492_34313708_7 . . . 713
21.156 so_7766087_7766087_2 . . . 713
21.157 so_7766087_18078800_0 . . . 713
21.158 so_38506473_38506473_0 . . . 713
21.159 so_194634_12399518_6 . . . 713
21.160 so_25169915_25171213_0 . . . 713
21.161 so_28418145_28418297_0 . . . 713
21.162 so_31910794_31910836_2 . . . 713
21.163 so_10570534_10570779_2 . . . 713
21.164 so_33124479_33124747_4 . . . 713
21.165 so_21110558_21110850_3 . . . 713
21.166 so_39289267_39289311_2 . . . 713
21.167 so_39289267_39289311_5 . . . 713
21.168 so_34646316_34646375_0 . . . 713
21.169 so_13792648_13792845_1 . . . 713
21.170 so_38874050_38874356_3 . . . 713
21.171 so_29190042_29190538_1 . . . 713
21.172 so_36936908_36936985_1 . . . 713
21.173 so_25068637_25068713_2 . . . 713
21.174 so_27551900_27552205_1 . . . 713
21.175 so_33657050_33675281_3 . . . 713
21.176 so_33657050_33675281_1 . . . 713
21.177 so_31910794_31910836_7 . . . 713
21.178 so_31910794_31910836_0 . . . 713
21.179 so_39289267_39289311_1 . . . 713
21.180 so_25068637_25068713_0 . . . 713
21.181 so_39200665_39200666_5 . . . 713
21.182 so_34313492_34313708_0 . . . 713
21.183 so_13597170_13599511_7 . . . 714
21.184 so_13574280_13575039_0 . . . 714
21.185 so_37074905_37075341_0 . . . 714
21.186 so_12285060_12285088_2 . . . 714
21.187 so_16629743_16629824_4 . . . 715
21.188 so_28142011_28142438_0 . . . 715
21.189 so_28142011_28142438_1 . . . 716
21.190 so_6285150_6285205_0 . . . 716
21.191 so_45934295_45934606_3 . . . 718
21.192 so_45934295_45934606_4 . . . 718
21.193 so_44033479_44033526_0 . . . 718
21.194 so_28953540_28953566_4 . . . 719

21.195 so_18370694_18371156_8 . . . 719
21.196 so_15050850_15050940_0 . . . 719
21.197 so_16629743_16629824_1 . . . 721
21.198 so_38985345_38985404_0 . . . 721
21.199 so_14962228_14962311_0 . . . 721
21.200 so_7554039_7554136_2 . . . 721
21.201 so_34313492_34313492_0 . . . 721
21.202 so_11119588_11119665_3 . . . 721
21.203 so_11209311_11209576_4 . . . 721
21.204 so_9057119_9057119_2 . . . 721
21.205 so_9057119_9057119_3 . . . 721
21.206 so_9057119_9057119_4 . . . 721
21.207 so_10187283_10187622_0 . . . 721
21.208 so_42062080_42062141_3 . . . 721
21.209 so_9057119_9057119_5 . . . 721
21.210 so_42062080_42062141_2 . . . 722
21.211 so_42062080_42062141_5 . . . 722
21.212 so_20593878_20593934_1 . . . 724
21.213 so_26344045_26344075_1 . . . 724
21.214 so_20593878_20593934_2 . . . 724
21.215 so_20593878_20593934_0 . . . 724
21.216 so_26344045_26344075_2 . . . 724
21.217 so_42062080_42062141_1 . . . 724
21.218 so_14591203_14591203_0 . . . 729
21.219 so_8766641_8766753_0 . . . 729
21.220 so_16067127_16067283_0 . . . 729
21.221 so_32298240_32298449_2 . . . 729
21.222 so_21809838_21818503_6 . . . 729
21.223 so_21809838_21818503_4 . . . 729
21.224 so_21809838_21818503_2 . . . 729
21.225 so_21809838_21818503_7 . . . 729
21.226 so_2960596_11145292_1 . . . 729
21.227 so_5695705_5695741_3 . . . 729
21.228 so_5695705_5695741_4 . . . 729
21.229 so_5695705_5695741_1 . . . 729
21.230 so_5695705_5696018_0 . . . 729
21.231 so_30444641_30444739_2 . . . 729
21.232 so_5695705_5695741_5 . . . 729
21.233 so_5695705_5695741_6 . . . 729
21.234 so_26818807_26818912_0 . . . 729
21.235 so_32298240_32298449_3 . . . 730
21.236 so_32298240_32298449_0 . . . 730
21.237 so_32298240_32298411_0 . . . 730
21.238 so_39704088_39704263_0 . . . 730
21.239 so_12152167_12152466_2 . . . 730
21.240 so_16468189_16473878_1 . . . 730
21.241 so_32298240_32298411_1 . . . 730
21.242 Numeric conversions . . . 730
21.243 so_6407814_6408104_0 . . . 730
21.244 so_38689286_38692249_0 . . . 730
21.245 so_39900952_39913976_2 . . . 730

21.246 so_39900952_39913976_0 . . . 730
21.247 so_39900952_39913976_1 . . . 730
21.248 so_24504269_24505925_0 . . . 730
21.249 so_38689286_38692249_6 . . . 730
21.250 so_39900952_39913976_3 . . . 731
21.251 so_10664699_10666695_0 . . . 731
21.252 so_9804853_9805164_1 . . . 731
21.253 so_13268608_13268608_1 . . . 731
21.254 so_9804853_9805106_0 . . . 731
21.255 so_4434140_4690988_0 . . . 731
21.256 so_28997175_28997175_2 . . . 731
21.257 so_22074295_22076708_0 . . . 731
21.258 Hash support . . . 732
21.259 Suffix for `basic_string` literals . . . 732
21.260 so_32116430_32118790_1 . . . 732
21.261 Null-terminated sequence utilities . . . 733
21.262 so_17504565_17504565_5 . . . 733
21.263 so_7131858_7132065_1 . . . 733
21.264 so_17504565_17504565_6 . . . 733
21.265 so_7131858_7132065_2 . . . 733
21.266 so_37496787_37496963_2 . . . 733
21.267 so_26873400_26873400_2 . . . 733
21.268 so_8380805_8380835_1 . . . 733
21.269 so_38689286_38692249_1 . . . 734

22 Localization library **737**
22.1 General . . . 737
22.2 Header `<locale>` synopsis . . . 737
22.3 Locales . . . 738
22.4 so_15168543_15168543_0 . . . 739
22.5 so_14719330_14719384_1 . . . 740
22.6 so_17779660_17779731_1 . . . 740
22.7 so_22647584_22647887_0 . . . 742
22.8 so_22647584_22647887_1 . . . 742
22.9 so_17779660_17779731_3 . . . 742
22.10 so_17779660_17779731_2 . . . 742
22.11 so_20409903_20410118_0 . . . 742
22.12 so_20409903_20410118_1 . . . 742
22.13 so_20409903_20410104_1 . . . 742
22.14 so_8285818_8371534_0 . . . 742
22.15 so_5750884_5751016_3 . . . 743
22.16 so_5750884_5751016_4 . . . 743
22.17 so_46364608_46364980_0 . . . 743
22.18 so_46364608_46364980_1 . . . 743
22.19 so_46364608_46364980_2 . . . 743
22.20 so_11190072_11190072_1 . . . 743
22.21 so_17681882_17682059_2 . . . 745
22.22 so_17681882_17682059_3 . . . 745
22.23 so_5750884_5751016_5 . . . 745
22.24 so_5750884_5751016_7 . . . 745
22.25 so_5750884_5751016_8 . . . 745

22.26 so_9416926_9416926_1 . . . 746
22.27 so_7131858_7132065_6 . . . 746
22.28 so_5849059_5849247_0 . . . 746
22.29 so_165723_168103_1 . . . 746
22.30 so_27090093_27090093_1 . . . 748
22.31 Standard `locale` categories . . . 750
22.32 so_29827028_29827144_0 . . . 756
22.33 so_17681882_17687578_1 . . . 757
22.34 so_17823693_17823693_1 . . . 757
22.35 so_621635_621655_1 . . . 758
22.36 so_621635_621655_2 . . . 758
22.37 so_621635_621655_3 . . . 758
22.38 so_1509277_1509474_1 . . . 758
22.39 so_11086183_11086361_0 . . . 758
22.40 so_41531492_41531492_2 . . . 759
22.41 so_41531492_41531492_1 . . . 759
22.42 so_8234888_8235021_0 . . . 759
22.43 so_19357027_19436896_1 . . . 760
22.44 so_41531492_41531492_3 . . . 760
22.45 so_28294129_28975080_1 . . . 761
22.46 so_24689378_24689378_3 . . . 764
22.47 so_13378989_13379959_1 . . . 764
22.48 so_13378989_13379959_2 . . . 764
22.49 so_6399615_19523324_1 . . . 764
22.50 so_6399615_19523324_2 . . . 764
22.51 so_35979062_35979551_2 . . . 764
22.52 so_35979062_35979551_1 . . . 764
22.53 so_24689378_24689378_4 . . . 764
22.54 so_24689378_24689378_5 . . . 764
22.55 so_24689378_24689378_6 . . . 764
22.56 so_24689378_24689378_8 . . . 764
22.57 so_24689378_24689549_1 . . . 764
22.58 so_24689378_24689378_7 . . . 764
22.59 so_24689378_24689549_0 . . . 764
22.60 so_36377836_39159956_0 . . . 764
22.61 so_13196528_13197147_0 . . . 764
22.62 so_13196528_13197252_4 . . . 764
22.63 so_23374095_23374456_1 . . . 764
22.64 so_23374095_23374456_2 . . . 764
22.65 so_19725070_19727956_0 . . . 764
22.66 so_22405031_22405182_2 . . . 764
22.67 so_22315683_22315876_0 . . . 764
22.68 so_22315683_22315876_1 . . . 764
22.69 so_22315683_22315876_5 . . . 764
22.70 so_22315683_22315876_6 . . . 764
22.71 so_22315683_22315876_4 . . . 764
22.72 so_22315683_22315876_2 . . . 764
22.73 so_22315683_22315876_7 . . . 764
22.74 so_22315683_22315876_3 . . . 764
22.75 so_13045693_13046160_2 . . . 764
22.76 so_13045693_13046160_1 . . . 764

22.77 so_13045693_13046160_0 . . . 764
22.78 so_10488176_10488269_0 . . . 764
22.79 so_32945923_32946596_2 . . . 764
22.80 so_32945923_32946596_3 . . . 764
22.81 so_32945923_32946596_4 . . . 764
22.82 so_32945923_32946596_6 . . . 764
22.83 so_32945923_32946596_1 . . . 764
22.84 so_24879216_24879341_0 . . . 766
22.85 so_10825743_10825885_2 . . . 766
22.86 so_10825743_10825885_1 . . . 766
22.87 so_10825743_10825885_0 . . . 766
22.88 so_24219807_24220637_0 . . . 766
22.89 so_9780087_9781003_2 . . . 766
22.90 so_9780087_9781003_1 . . . 766
22.91 so_16888145_16888979_2 . . . 768
22.92 so_16888145_16888979_4 . . . 768
22.93 so_26979592_26979891_0 . . . 768
22.94 so_34622698_34623467_0 . . . 768
22.95 so_29720288_29720461_4 . . . 768
22.96 so_8424260_8425199_4 . . . 768
22.97 so_8424260_8425199_0 . . . 768
22.98 so_8424260_8425199_2 . . . 768
22.99 so_15647220_15647278_1 . . . 768
22.100 so_28311795_28311795_1 . . . 768
22.101 so_20380274_20382471_1 . . . 768
22.102 so_20380274_20380274_0 . . . 768
22.103 so_37861536_37862742_3 . . . 776
22.104 so_37861536_37862742_1 . . . 776
22.105 so_37861536_37862742_0 . . . 776
22.106 so_14079108_14079538_0 . . . 782
22.107 so_14079108_14079538_1 . . . 784
22.108 Standard code conversion facets . . . 790
22.109 so_38770680_38772162_1 . . . 791
22.110 so_38770680_38770680_1 . . . 791
22.111 so_38770680_38772162_2 . . . 791
22.112 so_38770680_38772162_4 . . . 791
22.113 so_38770680_38770680_2 . . . 791
22.114 so_1509277_1509474_2 . . . 791
22.115 so_5666899_5667833_0 . . . 791
22.116 so_1509277_1509474_3 . . . 791
22.117 so_1509277_1509474_4 . . . 791
22.118 C library locales . . . 792

23 Containers library **793**
23.1 General . . . 793
23.2 Container requirements . . . 793
23.3 so_37710975_37712216_0 . . . 793
23.4 so_33144419_33144715_0 . . . 793
23.5 so_4010937_4010943_0 . . . 793
23.6 so_10770489_10770661_0 . . . 793
23.7 so_41290827_41290827_0 . . . 793

23.8 so_9552858_9553889_0 . . . 793
23.9 so_25670131_25682482_0 . . . 793
23.10 so_28050760_28052844_0 . . . 793
23.11 so_22440882_22528423_0 . . . 793
23.12 so_46541544_46543078_1 . . . 793
23.13 so_32983539_32983625_0 . . . 793
23.14 so_28201982_28203110_0 . . . 793
23.15 so_43340050_43352552_0 . . . 793
23.16 so_37663859_37664484_0 . . . 793
23.17 so_26852381_26852607_0 . . . 793
23.18 so_26852381_26852381_0 . . . 793
23.19 so_19532448_19535939_1 . . . 793
23.20 so_46409188_46409188_8 . . . 793
23.21 so_25127204_25127444_0 . . . 793
23.22 so_14674289_14674289_0 . . . 793
23.23 so_6169125_6169172_1 . . . 793
23.24 so_9746621_9746668_0 . . . 793
23.25 so_27830323_27831380_8 . . . 793
23.26 so_21458864_21459070_0 . . . 793
23.27 so_1115356_1115415_1 . . . 793
23.28 so_21314793_21319899_0 . . . 793
23.29 so_17796200_17796238_0 . . . 793
23.30 so_26660618_26660710_0 . . . 793
23.31 so_35680976_35747288_0 . . . 793
23.32 so_40609938_40610189_1 . . . 793
23.33 so_38280418_38280562_1 . . . 793
23.34 so_36475497_36478025_0 . . . 793
23.35 so_36475497_36475910_10 . . . 793
23.36 so_36475497_36475910_8 . . . 793
23.37 so_27508872_27509330_0 . . . 793
23.38 so_16900498_16900842_5 . . . 793
23.39 so_16900498_16900842_2 . . . 793
23.40 so_16900498_16900842_4 . . . 793
23.41 so_16900498_16900842_1 . . . 793
23.42 so_12332772_12332978_0 . . . 793
23.43 so_17730689_17731528_0 . . . 793
23.44 so_17730689_17731528_1 . . . 793
23.45 so_44033479_44033560_0 . . . 793
23.46 so_16422486_16422594_1 . . . 793
23.47 so_20479507_20479633_0 . . . 793
23.48 so_12608360_12608360_0 . . . 793
23.49 so_12997600_12997720_0 . . . 793
23.50 so_21568072_21568133_2 . . . 793
23.51 so_21568072_21568133_1 . . . 793
23.52 so_26472434_26473160_2 . . . 793
23.53 so_35390835_35390954_1 . . . 793
23.54 so_16927259_16927339_0 . . . 793
23.55 so_5725693_5725744_0 . . . 795
23.56 so_33843939_33843939_0 . . . 795
23.57 so_14592215_14592233_0 . . . 795
23.58 so_17796200_17796238_1 . . . 795

23.59 so_44767368_44767477_1 . . . 795
23.60 so_15599498_15599515_3 . . . 795
23.61 so_16781915_16781927_4 . . . 795
23.62 so_1432216_1434278_1 . . . 795
23.63 so_38214379_38214522_0 . . . 795
23.64 so_10793404_10793474_0 . . . 795
23.65 so_17730689_17731528_2 . . . 795
23.66 so_33655055_33655055_0 . . . 795
23.67 so_36506721_36506773_0 . . . 795
23.68 so_25783417_25783872_1 . . . 796
23.69 so_35390835_35392987_1 . . . 796
23.70 so_27830323_27831380_10 . . . 796
23.71 so_27830323_27831380_2 . . . 796
23.72 so_6322970_6323091_0 . . . 796
23.73 so_4732084_4732274_0 . . . 796
23.74 so_39961702_39961806_1 . . . 796
23.75 so_46409188_46409188_7 . . . 796
23.76 so_22995550_22995687_0 . . . 796
23.77 so_22384231_22384526_0 . . . 796
23.78 so_11703643_11703670_0 . . . 796
23.79 so_41854519_41854804_1 . . . 796
23.80 so_14674289_14674289_1 . . . 796
23.81 so_9727556_9727556_0 . . . 796
23.82 so_15851888_15851978_0 . . . 796
23.83 so_4898618_4899051_2 . . . 796
23.84 so_6322970_6323091_1 . . . 796
23.85 so_27735754_27735989_0 . . . 796
23.86 so_19611037_19612530_0 . . . 796
23.87 so_16894566_16894566_1 . . . 796
23.88 so_23758399_23758399_1 . . . 796
23.89 so_23758399_23758399_2 . . . 796
23.90 so_8190950_8190950_0 . . . 796
23.91 so_8190950_8190950_1 . . . 796
23.92 so_23757166_23757167_1 . . . 796
23.93 so_23757166_23757167_2 . . . 796
23.94 so_22322487_22323637_0 . . . 796
23.95 so_35679165_35679260_0 . . . 796
23.96 so_10373985_10374104_4 . . . 796
23.97 so_46874310_46874564_1 . . . 796
23.98 so_7829018_7829166_0 . . . 796
23.99 so_23754223_23755126_2 . . . 796
23.100 so_23754223_23755126_4 . . . 796
23.101 so_23754223_23755126_3 . . . 796
23.102 so_23754223_23755126_1 . . . 796
23.103 so_23754223_23755126_6 . . . 796
23.104 so_23754223_23755126_5 . . . 796
23.105 so_6390756_6390756_0 . . . 796
23.106 so_20479507_20479633_1 . . . 796
23.107 so_4124989_4125186_0 . . . 796
23.108 so_4124989_4125186_1 . . . 796
23.109 so_4124989_4125060_0 . . . 796

23.110 so_6322970_6323091_2 . . . 797
23.111 so_27453230_27453230_0 . . . 797
23.112 so_27453230_27453649_1 . . . 797
23.113 so_11699083_11699456_0 . . . 797
23.114 so_11699083_11699271_4 . . . 797
23.115 so_11699083_11699271_3 . . . 797
23.116 so_26902006_26904184_1 . . . 797
23.117 so_26902006_26904184_0 . . . 797
23.118 so_6642679_6643035_0 . . . 797
23.119 so_23764785_23764853_1 . . . 797
23.120 so_37262329_37262860_1 . . . 797
23.121 so_10373985_10374104_2 . . . 797
23.122 so_23962485_23962721_0 . . . 797
23.123 so_7902452_7902519_0 . . . 797
23.124 so_25347599_25348988_7 . . . 797
23.125 so_6210688_6212163_1 . . . 797
23.126 so_47438291_47438333_0 . . . 797
23.127 so_13428851_13428851_0 . . . 797
23.128 so_32190843_32190843_0 . . . 797
23.129 so_8065976_8066032_0 . . . 797
23.130 so_8065976_8066129_0 . . . 797
23.131 so_8899814_8899917_2 . . . 797
23.132 so_8899814_8899917_0 . . . 797
23.133 so_6438086_6442829_1 . . . 797
23.134 so_4124989_4125186_3 . . . 797
23.135 so_4124989_4125186_2 . . . 797
23.136 so_11488490_18858875_0 . . . 797
23.137 so_11021764_11022447_0 . . . 797
23.138 so_9006663_9006761_0 . . . 797
23.139 so_43564854_43565133_0 . . . 797
23.140 so_43564854_43565133_2 . . . 797
23.141 so_4380934_4381276_0 . . . 797
23.142 so_37262329_37262860_0 . . . 797
23.143 so_10687549_10687928_0 . . . 797
23.144 so_22253355_22253630_0 . . . 797
23.145 so_62340_62878_0 . . . 797
23.146 so_37571737_37571889_2 . . . 797
23.147 so_18428487_18428874_0 . . . 797
23.148 so_6438086_6442829_0 . . . 797
23.149 so_6438086_6438087_0 . . . 797
23.150 so_13912507_13912507_0 . . . 797
23.151 so_38071240_38071466_0 . . . 797
23.152 so_28640600_28640829_0 . . . 797
23.153 so_28640600_28640829_1 . . . 797
23.154 so_42074424_42074544_2 . . . 798
23.155 so_20603335_20603750_5 . . . 798
23.156 so_43407814_43408612_2 . . . 798
23.157 so_43407814_43408612_0 . . . 798
23.158 so_25903968_25904184_3 . . . 798
23.159 so_25903968_25904184_4 . . . 798
23.160 so_25903968_25904184_5 . . . 798

23.161 so_47474063_47475066_0 . . . 798
23.162 so_13812703_13813460_0 . . . 798
23.163 so_13812703_13813460_1 . . . 798
23.164 so_13812703_13813460_2 . . . 798
23.165 so_18900125_18901407_1 . . . 798
23.166 so_42736112_42736577_0 . . . 798
23.167 so_2577623_2577764_2 . . . 798
23.168 so_25330525_25330763_0 . . . 798
23.169 so_31206205_31206606_1 . . . 798
23.170 so_8467968_8467987_2 . . . 798
23.171 so_25803403_25803403_2 . . . 798
23.172 so_25803403_25803403_4 . . . 798
23.173 so_22860575_22883878_0 . . . 798
23.174 so_8654649_8655465_2 . . . 798
23.175 so_25198405_25198802_0 . . . 798
23.176 so_25198405_25198802_1 . . . 798
23.177 so_35901417_35901524_0 . . . 798
23.178 so_27830323_27831380_3 . . . 799
23.179 so_27830323_27831380_7 . . . 799
23.180 so_27221221_27221313_0 . . . 799
23.181 so_39040609_39044185_0 . . . 799
23.182 so_36475497_36492111_0 . . . 799
23.183 so_27471684_27472839_0 . . . 799
23.184 so_26472434_26473169_0 . . . 799
23.185 so_17667202_17667357_0 . . . 800
23.186 so_7225914_7225952_0 . . . 800
23.187 so_7225914_7226055_0 . . . 800
23.188 so_29381355_29381355_1 . . . 800
23.189 so_10833512_10833512_1 . . . 800
23.190 so_31993022_31993987_1 . . . 800
23.191 so_21408173_21408559_0 . . . 800
23.192 so_41387751_41388133_0 . . . 800
23.193 so_26867435_26867506_1 . . . 800
23.194 so_31131743_31131915_1 . . . 800
23.195 so_7455982_11144575_0 . . . 800
23.196 so_32461926_32462236_0 . . . 800
23.197 so_45793411_45794804_5 . . . 800
23.198 so_13132075_13132223_1 . . . 800
23.199 so_13955634_13955871_2 . . . 800
23.200 so_15067160_15067564_0 . . . 800
23.201 so_15067160_15067273_0 . . . 800
23.202 so_39037239_39037606_1 . . . 800
23.203 so_7416227_7416284_1 . . . 800
23.204 so_7225914_7226055_1 . . . 800
23.205 so_29381355_29381355_2 . . . 800
23.206 so_10833512_10833512_2 . . . 800
23.207 so_33617421_33618459_0 . . . 800
23.208 so_39163532_39163724_1 . . . 800
23.209 so_31993022_31993987_2 . . . 800
23.210 so_21408173_21408559_1 . . . 800
23.211 so_12931787_12938588_0 . . . 800

23.212 so_41387751_41388133_1 . . . 800
23.213 so_11019615_11020907_0 . . . 800
23.214 so_31131743_31131915_2 . . . 800
23.215 so_29226578_29226685_0 . . . 800
23.216 so_32461926_32462236_1 . . . 800
23.217 so_45793411_45794804_6 . . . 800
23.218 so_7372258_7372354_4 . . . 800
23.219 so_13955634_13955871_0 . . . 800
23.220 so_32815753_32815910_0 . . . 800
23.221 so_29381355_29381355_3 . . . 800
23.222 so_10833512_10833512_3 . . . 800
23.223 so_11019615_11020907_1 . . . 800
23.224 so_13955634_13955871_1 . . . 800
23.225 so_33843939_33844226_0 . . . 800
23.226 so_43564854_43565004_1 . . . 800
23.227 so_9184280_9184285_1 . . . 800
23.228 so_15008521_15008594_1 . . . 800
23.229 so_3738362_3738416_0 . . . 800
23.230 so_3738362_3738416_2 . . . 800
23.231 so_3738362_3738416_1 . . . 800
23.232 so_18788780_18789407_0 . . . 801
23.233 so_28456934_28457111_0 . . . 801
23.234 so_21786970_21787321_0 . . . 801
23.235 so_9215391_9215695_0 . . . 801
23.236 so_43564854_43565004_2 . . . 801
23.237 so_14094408_14094408_0 . . . 801
23.238 so_16616253_16616253_3 . . . 801
23.239 so_16616253_16616253_4 . . . 801
23.240 so_26902006_26904184_2 . . . 801
23.241 so_16174239_16174353_0 . . . 801
23.242 so_43550191_43550708_0 . . . 801
23.243 so_45079320_45079923_0 . . . 801
23.244 so_15599498_15599511_0 . . . 801
23.245 so_15142603_15142704_1 . . . 801
23.246 so_19826376_19826645_2 . . . 801
23.247 so_28097419_28097899_0 . . . 801
23.248 so_28097419_28097899_1 . . . 801
23.249 so_9215391_9215695_1 . . . 801
23.250 so_9046946_9047060_0 . . . 801
23.251 so_8654649_8655465_4 . . . 801
23.252 so_15716665_15716841_1 . . . 801
23.253 so_23761273_23761589_2 . . . 801
23.254 so_10218223_10220042_0 . . . 801
23.255 so_14647007_14647210_0 . . . 801
23.256 so_38570267_38570582_1 . . . 801
23.257 so_2998452_2998452_0 . . . 803
23.258 so_6230350_24251570_0 . . . 803
23.259 so_28822474_28822474_0 . . . 803
23.260 so_43564854_43565133_1 . . . 803
23.261 so_4713131_4713162_0 . . . 803
23.262 so_42536333_42536449_0 . . . 803

23.263 so_42536333_42536449_2 . . . 803
23.264 so_19604660_19604731_1 . . . 803
23.265 so_24813219_24814591_4 . . . 803
23.266 so_24813219_24814591_3 . . . 803
23.267 so_24813219_24814591_0 . . . 803
23.268 so_26073740_26073740_0 . . . 803
23.269 so_11155643_12432482_1 . . . 803
23.270 so_14035520_14035612_0 . . . 803
23.271 so_14035520_14035612_2 . . . 803
23.272 so_45847787_45848470_8 . . . 803
23.273 so_45847787_45848470_6 . . . 803
23.274 so_13792413_13792413_2 . . . 803
23.275 so_13792413_13792413_0 . . . 803
23.276 so_25918446_25918723_5 . . . 803
23.277 so_25918446_25918751_3 . . . 803
23.278 so_25918446_25918751_4 . . . 803
23.279 so_14035520_14035612_3 . . . 804
23.280 so_45847787_45848470_9 . . . 804
23.281 so_13792413_13792413_3 . . . 804
23.282 so_28529376_28529503_0 . . . 804
23.283 so_18788780_18789407_2 . . . 804
23.284 so_18788780_18789179_0 . . . 804
23.285 so_20603335_20603750_3 . . . 804
23.286 so_20603335_20603750_0 . . . 804
23.287 so_15599498_15599515_1 . . . 804
23.288 so_17260388_17260413_0 . . . 804
23.289 so_19826376_19826645_0 . . . 804
23.290 so_19941601_19941865_0 . . . 804
23.291 so_8335430_8338220_1 . . . 804
23.292 so_18900125_18901407_0 . . . 804
23.293 so_22988889_22989006_2 . . . 804
23.294 so_5322104_5322234_1 . . . 804
23.295 so_8013599_8014673_2 . . . 805
23.296 so_11809163_41424708_0 . . . 805
23.297 so_8833938_41424755_0 . . . 805
23.298 so_15761239_15761416_1 . . . 805
23.299 so_8013599_8014673_3 . . . 805
23.300 so_3349655_3349773_0 . . . 805
23.301 so_16815498_16815697_0 . . . 805
23.302 so_5989010_5989010_0 . . . 805
23.303 so_40572451_40573215_0 . . . 805
23.304 so_10968494_10968509_0 . . . 805
23.305 so_41180181_41180281_0 . . . 805
23.306 so_34430992_34431070_1 . . . 805
23.307 so_31952252_31962963_0 . . . 805
23.308 so_21697982_21698697_0 . . . 805
23.309 so_31051754_31051859_0 . . . 805
23.310 so_17328097_17328249_0 . . . 805
23.311 so_21785577_21785667_0 . . . 805
23.312 so_16261441_16261488_1 . . . 805
23.313 so_1380567_1380586_0 . . . 805

23.314 so_40572451_40573110_0 . . . 805
23.315 so_31035255_31035481_0 . . . 805
23.316 so_34430992_34431070_2 . . . 805
23.317 so_16870557_16870897_0 . . . 805
23.318 so_15701123_15701171_0 . . . 805
23.319 so_15880204_15880292_0 . . . 805
23.320 so_2643473_2643599_0 . . . 805
23.321 so_21013322_21013710_0 . . . 805
23.322 so_41360471_41360643_0 . . . 805
23.323 so_34582666_34582686_2 . . . 805
23.324 so_5337718_5337718_2 . . . 805
23.325 so_5849248_5849508_0 . . . 806
23.326 so_27868856_27869252_0 . . . 806
23.327 so_36475497_36475910_1 . . . 806
23.328 so_5849248_5849508_1 . . . 806
23.329 so_42036691_42036691_1 . . . 806
23.330 so_25557815_25557857_0 . . . 806
23.331 so_16781915_16781927_7 . . . 806
23.332 so_24185573_24186122_0 . . . 806
23.333 so_18277038_18277402_1 . . . 806
23.334 so_27868856_27869252_1 . . . 806
23.335 so_25931806_25932431_0 . . . 806
23.336 so_5973427_5973670_0 . . . 806
23.337 so_8719505_8719630_0 . . . 806
23.338 so_2038453_2038534_0 . . . 806
23.339 so_15716665_15716841_0 . . . 806
23.340 so_12283302_12283411_0 . . . 806
23.341 so_27221221_27221313_1 . . . 806
23.342 so_35444114_35445106_1 . . . 806
23.343 so_31623423_31667355_0 . . . 806
23.344 so_36475497_36475910_2 . . . 806
23.345 so_23214184_23214949_0 . . . 806
23.346 so_37184970_37185665_0 . . . 806
23.347 so_24156369_24156369_1 . . . 806
23.348 so_24156369_24795559_0 . . . 806
23.349 so_21617975_21618411_0 . . . 806
23.350 so_10261705_10261705_1 . . . 806
23.351 so_10261705_10261705_0 . . . 806
23.352 so_16781915_16781927_3 . . . 806
23.353 so_16781915_16781927_5 . . . 806
23.354 so_12607316_12609771_0 . . . 806
23.355 so_11859262_11859262_0 . . . 806
23.356 so_21786970_21787269_0 . . . 806
23.357 so_36214877_36215687_0 . . . 806
23.358 so_17016244_17016244_0 . . . 806
23.359 so_40502357_40505121_0 . . . 806
23.360 so_22676483_22676597_0 . . . 806
23.361 so_25809873_25810115_1 . . . 806
23.362 so_25809873_25810115_2 . . . 806
23.363 so_47483791_47484559_0 . . . 806
23.364 so_32722789_32723582_0 . . . 806

23.365 so_3639741_29352251_0 . . . 806
23.366 so_34856965_34857406_5 . . . 806
23.367 so_16534931_16534996_0 . . . 806
23.368 so_16534931_16535030_0 . . . 806
23.369 so_10150154_10150172_1 . . . 806
23.370 so_28097419_28097899_3 . . . 806
23.371 so_17400137_17403467_4 . . . 806
23.372 so_19347862_19348825_2 . . . 806
23.373 so_10043716_10046261_7 . . . 806
23.374 so_10043716_10046261_2 . . . 806
23.375 so_12424329_12424421_1 . . . 806
23.376 so_12424329_12424421_0 . . . 806
23.377 so_38554741_38554862_1 . . . 806
23.378 so_28822474_28822474_1 . . . 806
23.379 so_4344137_4344146_1 . . . 806
23.380 so_17552191_17553165_1 . . . 806
23.381 so_17552191_17553066_0 . . . 806
23.382 so_17552191_17553165_2 . . . 806
23.383 so_45847787_45847989_0 . . . 806
23.384 so_20328242_20328676_0 . . . 806
23.385 so_44530264_44530264_5 . . . 806
23.386 so_20735255_20735326_1 . . . 806
23.387 so_28235632_28235632_3 . . . 806
23.388 so_28235632_28235632_1 . . . 806
23.389 so_15414541_15414926_0 . . . 812
23.390 so_47576004_47576277_0 . . . 812
23.391 so_9006663_9006761_1 . . . 812
23.392 so_2364662_2364680_0 . . . 812
23.393 so_446205_446212_0 . . . 812
23.394 so_9756775_9756823_1 . . . 812
23.395 so_14492894_14493258_0 . . . 812
23.396 so_1068557_1069200_0 . . . 812
23.397 so_1801251_1801384_1 . . . 812
23.398 so_1432216_1434278_0 . . . 812
23.399 so_1432216_1432613_1 . . . 812
23.400 so_33881285_33881710_0 . . . 812
23.401 so_3856771_3856878_0 . . . 812
23.402 so_23798011_23798087_0 . . . 812
23.403 so_12681780_12681823_0 . . . 812
23.404 so_1636578_1636609_0 . . . 812
23.405 so_41436045_41436134_0 . . . 812
23.406 so_2874441_2874533_0 . . . 812
23.407 so_24082294_24082414_0 . . . 812
23.408 so_47280380_47280517_0 . . . 812
23.409 so_9886490_9886696_1 . . . 812
23.410 so_9886490_9886696_0 . . . 812
23.411 so_9886490_9886852_0 . . . 812
23.412 so_26867435_26869014_0 . . . 812
23.413 so_17601722_17602136_2 . . . 812
23.414 so_27889722_27889810_0 . . . 812
23.415 so_10033567_10033625_1 . . . 812

23.416 so_4767228_4767273_0 . . . 812
23.417 so_18428487_18428874_1 . . . 812
23.418 so_36115878_36116463_0 . . . 812
23.419 so_13278093_13278168_2 . . . 812
23.420 so_516007_516101_0 . . . 812
23.421 so_4343220_4343287_0 . . . 812
23.422 so_3349655_3349714_0 . . . 812
23.423 so_11809163_41424708_1 . . . 812
23.424 so_8833938_41424755_1 . . . 812
23.425 so_10968494_10968509_1 . . . 812
23.426 so_16827122_16827409_0 . . . 812
23.427 so_16827122_16827409_1 . . . 812
23.428 so_45218398_45218608_0 . . . 812
23.429 so_20317413_20318388_11 . . . 812
23.430 so_20317413_20317413_0 . . . 812
23.431 so_35525777_35525806_0 . . . 812
23.432 so_27763240_27763339_0 . . . 812
23.433 so_20735255_20735326_2 . . . 812
23.434 so_17312619_17312804_1 . . . 812
23.435 so_17251629_17251909_0 . . . 813
23.436 so_17312619_17312804_2 . . . 813
23.437 so_17312619_17312804_3 . . . 813
23.438 so_25670131_25682482_1 . . . 813
23.439 so_40454514_40454850_1 . . . 813
23.440 so_10405030_10405129_0 . . . 813
23.441 so_45014513_45014642_0 . . . 813
23.442 so_19720064_19851499_0 . . . 813
23.443 so_25556687_25556781_0 . . . 813
23.444 so_17783622_17783927_0 . . . 813
23.445 so_25556687_25556781_1 . . . 813
23.446 so_35444114_35445106_0 . . . 813
23.447 so_31772406_31773038_0 . . . 813
23.448 so_45014513_45014642_2 . . . 813
23.449 so_23706171_23706272_0 . . . 813
23.450 so_20803575_20815574_0 . . . 813
23.451 so_27606324_27606378_0 . . . 813
23.452 so_41404733_41404983_0 . . . 813
23.453 so_22412150_22412688_0 . . . 813
23.454 so_22412150_22412688_1 . . . 813
23.455 so_34582666_34582686_0 . . . 813
23.456 so_28960167_28960335_0 . . . 813
23.457 so_25536376_25537837_0 . . . 813
23.458 so_38415914_38416985_0 . . . 813
23.459 so_40454514_40454850_3 . . . 814
23.460 so_21518704_21519560_0 . . . 814
23.461 so_24991327_24991657_0 . . . 814
23.462 so_17241750_17241943_0 . . . 814
23.463 so_22412150_22412688_2 . . . 814
23.464 so_34582666_34582686_1 . . . 814
23.465 so_39868640_39869063_2 . . . 814
23.466 so_18292433_18293070_0 . . . 814

23.467 so_13730470_40892857_1 . . . 814
23.468 so_36242103_36242182_0 . . . 814
23.469 so_18428487_18428874_2 . . . 814
23.470 so_8664295_8664429_0 . . . 814
23.471 so_44690986_44691066_0 . . . 814
23.472 so_12901346_12901404_0 . . . 814
23.473 so_23706171_23706272_1 . . . 814
23.474 so_32247436_32247530_0 . . . 814
23.475 so_704780_705611_1 . . . 814
23.476 so_39200665_39200666_16 . . . 814
23.477 so_29977259_29977327_0 . . . 814
23.478 so_11295264_11295394_0 . . . 814
23.479 so_25670131_25682482_2 . . . 814
23.480 so_25670131_25682482_3 . . . 814
23.481 so_26874140_26874374_0 . . . 814
23.482 so_3105798_26985436_1 . . . 814
23.483 so_16781915_16781927_1 . . . 814
23.484 so_25299888_25299888_1 . . . 814
23.485 so_30883069_30883224_0 . . . 814
23.486 so_46046924_46046975_1 . . . 814
23.487 so_41186954_41187238_0 . . . 814
23.488 so_11859262_11859390_0 . . . 814
23.489 so_11859262_11859390_1 . . . 814
23.490 so_25600484_25600508_0 . . . 814
23.491 so_30187915_30188026_2 . . . 814
23.492 so_37344362_37344684_1 . . . 814
23.493 so_17016244_17018742_1 . . . 814
23.494 so_22412150_22412688_4 . . . 814
23.495 so_22412150_22412696_2 . . . 814
23.496 so_22412150_22412688_3 . . . 814
23.497 so_34238806_34238912_1 . . . 814
23.498 so_28097419_28097899_4 . . . 814
23.499 so_4780605_4780605_0 . . . 814
23.500 so_8467968_8467987_1 . . . 814
23.501 so_10043716_10046261_3 . . . 814
23.502 so_27699118_27699462_2 . . . 814
23.503 so_13730470_13730864_3 . . . 814
23.504 so_13730470_13730864_0 . . . 814
23.505 so_13730470_13730864_2 . . . 814
23.506 so_20574035_20574126_1 . . . 814
23.507 so_17333194_17333648_1 . . . 814
23.508 so_38554741_38554862_2 . . . 814
23.509 so_15188747_38409783_0 . . . 814
23.510 so_36617248_36617331_0 . . . 814
23.511 so_12750531_12750740_0 . . . 814
23.512 so_42363040_42363605_2 . . . 814
23.513 so_28949774_28949939_7 . . . 814
23.514 so_24227443_24227514_0 . . . 814
23.515 so_44530264_45053735_1 . . . 814
23.516 so_44530264_44530264_1 . . . 814
23.517 so_44530264_44530264_3 . . . 814

23.518 so_20105329_20105367_1 . . . 814
23.519 so_16068151_16068170_1 . . . 814
23.520 so_17681779_17681779_20 . . . 814
23.521 so_37043401_37055970_0 . . . 814
23.522 so_6437089_6441763_0 . . . 822
23.523 so_2772742_2772880_0 . . . 822
23.524 so_30883069_30883224_3 . . . 822
23.525 so_30883069_30883224_1 . . . 822
23.526 so_30883069_30883224_2 . . . 822
23.527 so_18818909_18818960_0 . . . 822
23.528 so_25536376_25539172_0 . . . 822
23.529 so_36167764_36168090_0 . . . 822
23.530 so_36167764_36168037_1 . . . 822
23.531 so_16781886_16782536_0 . . . 823
23.532 so_13730470_40892857_0 . . . 823
23.533 so_38468844_38468844_3 . . . 823
23.534 so_18428487_18428874_3 . . . 823
23.535 so_16781886_16782536_1 . . . 823
23.536 so_31112852_31113618_0 . . . 823
23.537 so_7474126_7475106_0 . . . 823
23.538 so_9886490_9886852_1 . . . 823
23.539 so_18428487_18428874_4 . . . 823
23.540 Sequence containers . . . 823
23.541 so_21804976_21807738_0 . . . 823
23.542 so_12743408_12743408_0 . . . 823
23.543 so_6632915_6632937_0 . . . 826
23.544 so_47120994_47121087_0 . . . 826
23.545 so_16962973_16966072_0 . . . 826
23.546 so_25173462_25173547_0 . . . 826
23.547 so_29150369_29150503_1 . . . 826
23.548 so_29150369_29150584_0 . . . 826
23.549 so_29150369_29150503_0 . . . 826
23.550 so_6632915_6632937_1 . . . 826
23.551 so_18155239_18155370_0 . . . 826
23.552 so_19103244_19103585_0 . . . 826
23.553 so_19103244_19103585_1 . . . 826
23.554 so_39548254_39548345_0 . . . 826
23.555 so_22518680_22518680_0 . . . 826
23.556 so_32521974_32521974_0 . . . 826
23.557 so_32521974_32521974_2 . . . 826
23.558 so_31278377_31278474_4 . . . 826
23.559 so_31278377_31278474_5 . . . 826
23.560 so_32576388_32576468_1 . . . 826
23.561 so_32576388_32576468_0 . . . 826
23.562 so_13905200_28936701_0 . . . 826
23.563 so_29593597_29593652_0 . . . 826
23.564 so_21763002_21763025_1 . . . 826
23.565 so_21763002_21763025_3 . . . 826
23.566 so_39376813_39379359_0 . . . 826
23.567 so_28541488_28542142_1 . . . 826
23.568 so_28541488_28542142_0 . . . 826

23.569 so_19318482_19319374_1 . . . 826
23.570 so_19318482_19319374_3 . . . 826
23.571 so_3674247_3674759_0 . . . 826
23.572 so_25173462_25173547_2 . . . 826
23.573 so_25173462_25173547_1 . . . 826
23.574 so_40238484_40238583_1 . . . 826
23.575 so_28229382_28229382_0 . . . 826
23.576 so_41853089_41853089_0 . . . 826
23.577 so_6632915_6632937_2 . . . 827
23.578 so_32576388_32576468_3 . . . 827
23.579 so_39376813_39377033_0 . . . 827
23.580 so_18105139_18105697_0 . . . 827
23.581 so_22613991_22613991_0 . . . 827
23.582 so_24152435_24152435_0 . . . 828
23.583 so_24152435_24152435_1 . . . 828
23.584 so_24152435_24152435_2 . . . 828
23.585 so_8924088_8924604_0 . . . 828
23.586 so_24152435_24152435_3 . . . 828
23.587 so_15310989_15311159_2 . . . 828
23.588 so_15310989_15311159_1 . . . 828
23.589 so_42036691_42036893_1 . . . 829
23.590 so_37706380_37706473_1 . . . 829
23.591 so_22306949_22307025_1 . . . 829
23.592 so_2974470_2974470_3 . . . 829
23.593 so_15191741_15191850_0 . . . 831
23.594 so_9727556_9727556_1 . . . 831
23.595 so_10017607_10018555_1 . . . 831
23.596 so_21004879_21005208_1 . . . 831
23.597 so_25918446_25918751_1 . . . 831
23.598 so_15047809_15047846_1 . . . 832
23.599 so_14895428_14895493_1 . . . 832
23.600 so_17428375_17429093_0 . . . 832
23.601 so_22863804_22863899_1 . . . 832
23.602 so_22863804_22863899_4 . . . 832
23.603 so_12259771_12259886_0 . . . 832
23.604 so_30937208_30937290_0 . . . 832
23.605 so_30937208_30937208_1 . . . 832
23.606 so_30935840_30936261_10 . . . 832
23.607 so_28227959_28228015_0 . . . 832
23.608 so_37684035_37684098_0 . . . 832
23.609 so_46958273_46958961_1 . . . 832
23.610 so_29654206_29654899_4 . . . 832
23.611 so_17003335_17003379_0 . . . 832
23.612 so_39324192_39324473_0 . . . 832
23.613 so_14793086_14793119_0 . . . 832
23.614 so_10373985_10374104_1 . . . 832
23.615 so_10373985_10374104_0 . . . 832
23.616 so_17516311_17516311_0 . . . 832
23.617 so_25778492_25778602_0 . . . 832
23.618 so_7052677_7052863_0 . . . 832
23.619 so_913980_914037_1 . . . 832

23.620 so_14084475_14084616_0 . . . 832
23.621 so_27453230_27453649_0 . . . 832
23.622 so_11699083_11699456_5 . . . 832
23.623 so_11699083_11699456_3 . . . 832
23.624 so_46874310_46874564_0 . . . 832
23.625 so_36453394_36453605_2 . . . 832
23.626 so_36453394_36453605_3 . . . 832
23.627 so_40483483_40483483_1 . . . 832
23.628 so_17890219_17890265_2 . . . 832
23.629 so_22306949_22307027_2 . . . 832
23.630 so_8335430_8335430_1 . . . 832
23.631 so_31758025_31758082_3 . . . 832
23.632 so_33994190_33994288_0 . . . 832
23.633 so_29899282_29899835_0 . . . 832
23.634 so_1387647_1387747_1 . . . 833
23.635 so_10373985_10374104_3 . . . 833
23.636 so_18180127_18180414_0 . . . 833
23.637 so_25778492_25778602_1 . . . 833
23.638 so_37476160_37476318_1 . . . 833
23.639 so_37476160_37476263_2 . . . 833
23.640 so_37476160_37476318_0 . . . 833
23.641 so_37476160_37476318_2 . . . 833
23.642 so_28277982_30244945_2 . . . 833
23.643 so_28277982_28277982_1 . . . 833
23.644 so_28266382_28266883_3 . . . 833
23.645 so_28266382_28266883_4 . . . 833
23.646 so_11699083_11699456_4 . . . 833
23.647 so_11699083_11699456_6 . . . 833
23.648 so_28136080_28136248_0 . . . 833
23.649 so_28136080_28139567_1 . . . 833
23.650 so_28277982_28277982_2 . . . 833
23.651 so_28277982_28277982_5 . . . 833
23.652 so_29899282_29899835_3 . . . 833
23.653 so_42036691_42036893_0 . . . 833
23.654 so_15765631_15766477_0 . . . 833
23.655 so_27881869_27881899_0 . . . 833
23.656 so_24452530_24489607_0 . . . 836
23.657 so_17642800_17643322_0 . . . 837
23.658 so_18010848_18011413_3 . . . 837
23.659 so_18010848_18011413_4 . . . 837
23.660 so_913980_914037_0 . . . 837
23.661 so_18010848_18011413_7 . . . 838
23.662 so_29720288_29720461_5 . . . 838
23.663 so_44530264_44530264_6 . . . 839
23.664 so_44530264_44530264_2 . . . 839
23.665 so_16751734_16751908_3 . . . 839
23.666 so_44965206_44965206_3 . . . 839
23.667 so_14647007_14647540_0 . . . 839
23.668 so_44965206_44965206_5 . . . 839
23.669 so_44965206_44965206_9 . . . 840
23.670 so_6230350_24251570_1 . . . 840

23.671 so_11106313_11106408_0 . . . 840
23.672 so_4484900_24481485_5 . . . 840
23.673 so_5056973_5056994_0 . . . 841
23.674 so_14369499_14370750_1 . . . 841
23.675 so_3191790_3191977_0 . . . 841
23.676 so_12755837_12755902_0 . . . 841
23.677 so_4700052_4700073_3 . . . 841
23.678 so_14647007_14647210_1 . . . 841
23.679 so_22863804_22863899_5 . . . 844
23.680 so_22863804_22863899_2 . . . 844
23.681 so_22863804_22864073_0 . . . 844
23.682 so_12750531_12750695_2 . . . 845
23.683 so_13157164_13157393_3 . . . 845
23.684 so_13157164_13157393_4 . . . 845
23.685 so_15612700_15613125_0 . . . 846
23.686 so_25256969_25257307_1 . . . 846
23.687 so_15612700_15613125_1 . . . 846
23.688 so_13157164_13157393_5 . . . 846
23.689 so_13157164_13157393_6 . . . 846
23.690 so_13157164_13157210_0 . . . 846
23.691 so_30154882_30154999_0 . . . 846
23.692 so_23681954_23682279_4 . . . 847
23.693 so_23681954_23682279_3 . . . 847
23.694 so_23681954_23682279_5 . . . 847
23.695 so_26477016_26477051_0 . . . 847
23.696 so_4484900_24481485_7 . . . 847
23.697 so_39980854_39980889_1 . . . 848
23.698 so_8871607_8871647_0 . . . 848
23.699 so_37587309_37592118_0 . . . 848
23.700 so_14094408_14094408_1 . . . 848
23.701 so_247738_247902_0 . . . 848
23.702 so_247738_247764_0 . . . 848
23.703 so_32090595_32090711_0 . . . 848
23.704 so_32090595_32090697_1 . . . 848
23.705 so_32090595_32090697_2 . . . 848
23.706 so_24346232_24346324_0 . . . 848
23.707 so_5056973_5056994_1 . . . 848
23.708 so_28034635_28034669_2 . . . 848
23.709 so_22013697_22013788_1 . . . 848
23.710 so_35535911_35536737_0 . . . 848
23.711 so_3093451_3093470_0 . . . 848
23.712 so_19728012_19728048_0 . . . 848
23.713 so_22306949_22307025_3 . . . 848
23.714 so_33642579_33642597_1 . . . 848
23.715 so_12580628_12580789_0 . . . 848
23.716 so_12058838_12058871_0 . . . 848
23.717 so_666320_666508_0 . . . 848
23.718 so_1329415_1329442_1 . . . 848
23.719 so_849168_849190_1 . . . 848
23.720 so_16912880_16913043_0 . . . 848
23.721 so_21592196_21592253_1 . . . 848

23.722 so_5020076_5020268_3 . . . 848
23.723 so_23635310_23635310_0 . . . 848
23.724 so_23635310_23635362_3 . . . 848
23.725 so_3077748_3077767_0 . . . 848
23.726 so_36330536_36330913_1 . . . 848
23.727 so_34842454_34842591_3 . . . 848
23.728 so_39801898_39801988_0 . . . 848
23.729 so_39801898_39801982_1 . . . 848
23.730 so_18759692_18759792_0 . . . 848
23.731 so_15824119_15824568_0 . . . 848
23.732 so_35094545_35094731_0 . . . 848
23.733 so_4700052_4700073_1 . . . 848
23.734 so_15188747_15510314_0 . . . 848
23.735 so_28770352_28770613_1 . . . 848
23.736 so_15875994_15876040_1 . . . 848
23.737 so_20829860_20829969_0 . . . 848
23.738 so_6389521_6389769_0 . . . 848
23.739 so_2096571_2097179_0 . . . 848
23.740 so_22932984_22933125_0 . . . 848
23.741 so_46958273_46958961_0 . . . 848
23.742 so_7609169_7609294_0 . . . 848
23.743 so_37206756_37206971_0 . . . 848
23.744 so_30168069_30168311_1 . . . 848
23.745 so_32034115_32034260_0 . . . 848
23.746 so_62340_62840_0 . . . 848
23.747 so_29225448_29225695_1 . . . 848
23.748 so_46409188_46409188_6 . . . 848
23.749 so_25903968_25904184_0 . . . 850
23.750 so_25903968_25904184_1 . . . 850
23.751 so_23920595_23920678_0 . . . 851
23.752 so_21786970_21787293_0 . . . 851
23.753 so_23920595_23920678_1 . . . 851
23.754 so_2003266_2003695_0 . . . 851
23.755 so_9215391_9215695_2 . . . 851
23.756 so_9215391_9215391_0 . . . 851
23.757 so_9215391_9215809_0 . . . 851
23.758 so_15530321_15530585_1 . . . 851
23.759 so_1787966_1788037_0 . . . 851
23.760 so_27431060_27431789_0 . . . 851
23.761 so_5410035_5410129_0 . . . 851
23.762 so_37571737_37571889_0 . . . 851
23.763 so_41557421_41557463_1 . . . 851
23.764 so_46029612_46034235_0 . . . 851
23.765 so_26807356_26807444_0 . . . 851
23.766 so_26807356_26807506_0 . . . 851
23.767 so_15047809_15047846_0 . . . 851
23.768 so_15730992_15730993_0 . . . 851
23.769 so_14940511_14941147_1 . . . 851
23.770 so_5410035_5410229_1 . . . 851
23.771 so_47438291_47438449_1 . . . 851
23.772 so_18735910_18735953_0 . . . 851

23.773 so_9304894_9304933_5 . . . 851
23.774 so_28227959_28228015_2 . . . 851
23.775 so_10127603_10128180_0 . . . 851
23.776 so_27581754_27582018_0 . . . 851
23.777 so_27581754_27581998_0 . . . 851
23.778 so_9304894_9304933_6 . . . 851
23.779 so_13314638_13316655_4 . . . 851
23.780 so_13314638_13316655_2 . . . 851
23.781 so_14900104_14900222_0 . . . 851
23.782 so_2358516_2358540_0 . . . 851
23.783 so_10687549_10687928_1 . . . 851
23.784 so_20877837_20877982_0 . . . 851
23.785 so_24260717_24266689_0 . . . 851
23.786 so_27431060_27431789_2 . . . 851
23.787 so_27431060_27431789_3 . . . 851
23.788 so_12674749_12674927_0 . . . 851
23.789 so_5410035_5410129_1 . . . 851
23.790 so_22232403_22233386_1 . . . 851
23.791 so_37571737_37571889_1 . . . 851
23.792 so_9304894_9304933_8 . . . 851
23.793 so_16508509_16516338_0 . . . 851
23.794 so_6438086_22264195_0 . . . 851
23.795 so_6438086_22264195_1 . . . 851
23.796 so_39361878_39362416_0 . . . 851
23.797 so_39361878_39362463_1 . . . 851
23.798 so_29654206_29654899_2 . . . 851
23.799 so_45934295_45934613_0 . . . 851
23.800 so_1111078_1111311_0 . . . 851
23.801 so_2664051_2664051_0 . . . 851
23.802 so_25419851_25420131_3 . . . 852
23.803 so_36007044_36007537_3 . . . 852
23.804 so_39376813_39377268_0 . . . 852
23.805 so_38972443_38972840_1 . . . 852
23.806 so_27581754_27581754_0 . . . 852
23.807 so_43846969_43847210_0 . . . 852
23.808 so_41557421_41557463_3 . . . 852
23.809 so_14900104_14900172_0 . . . 852
23.810 so_15050632_15051314_2 . . . 852
23.811 so_27453230_27453230_1 . . . 852
23.812 so_16616253_16659850_1 . . . 852
23.813 so_16616253_16616253_5 . . . 852
23.814 so_14820835_14820933_0 . . . 852
23.815 so_31659968_31660313_0 . . . 852
23.816 so_1387647_1387747_0 . . . 852
23.817 so_1801251_1801384_0 . . . 852
23.818 so_17636690_17636736_0 . . . 852
23.819 so_27677294_27677551_1 . . . 852
23.820 so_46177840_46179994_2 . . . 852
23.821 so_18010848_18011413_10 . . . 852
23.822 so_23746042_23746114_0 . . . 852
23.823 so_46874310_46874444_0 . . . 852

23.824 so_12707444_12707588_2 . . . 852
23.825 so_12707444_12709211_2 . . . 852
23.826 so_27431060_27431789_1 . . . 852
23.827 so_5410035_5410229_2 . . . 852
23.828 so_19532448_19535309_0 . . . 852
23.829 so_19532448_19535939_0 . . . 852
23.830 so_35359169_35359296_1 . . . 852
23.831 so_35359169_35359225_0 . . . 852
23.832 so_22232403_22233386_0 . . . 852
23.833 so_22232403_22232447_0 . . . 852
23.834 so_13172888_13172888_0 . . . 852
23.835 so_37571737_37571889_3 . . . 852
23.836 so_37571737_37571873_0 . . . 852
23.837 so_22773559_22773644_0 . . . 852
23.838 so_46409188_46409188_5 . . . 852
23.839 so_34142490_34167803_0 . . . 852
23.840 so_26532884_26532884_1 . . . 852
23.841 so_26532884_26532884_0 . . . 852
23.842 so_20098454_20098536_3 . . . 852
23.843 so_20098454_20098536_4 . . . 852
23.844 so_20098454_20098536_2 . . . 852
23.845 so_16508509_16516338_1 . . . 852
23.846 so_5396634_5396710_0 . . . 852
23.847 so_18907605_18907887_1 . . . 852
23.848 so_35938329_35938555_1 . . . 852
23.849 so_45125134_45125235_1 . . . 852
23.850 so_27364958_27364958_0 . . . 852
23.851 so_11209311_11209425_0 . . . 852
23.852 so_11488490_18858875_2 . . . 852
23.853 so_39361878_39362463_0 . . . 852
23.854 so_40483483_40483483_2 . . . 852
23.855 so_43430407_43430605_1 . . . 852
23.856 so_10218223_10220042_1 . . . 852
23.857 so_15058363_15058395_0 . . . 852
23.858 so_27453230_27453230_2 . . . 852
23.859 so_16616253_16616253_6 . . . 852
23.860 so_18010848_18011413_11 . . . 852
23.861 so_10427659_10427814_0 . . . 852
23.862 so_32186277_32186340_0 . . . 852
23.863 so_11490242_11490242_1 . . . 853
23.864 so_11490242_11490727_0 . . . 853
23.865 so_15050632_15051314_1 . . . 853
23.866 so_24495889_24496183_0 . . . 853
23.867 so_43564854_43564854_0 . . . 853
23.868 so_31659968_31660313_1 . . . 853
23.869 so_62340_62878_1 . . . 853
23.870 so_17428375_17429093_1 . . . 853
23.871 so_12707444_12709211_4 . . . 853
23.872 so_12707444_12707588_5 . . . 853
23.873 so_31758025_31758068_1 . . . 853
23.874 so_24708480_24708610_0 . . . 853

23.875 so_25868107_25868340_0 . . . 853
23.876 so_25868107_25868107_0 . . . 853
23.877 so_29899282_29899835_1 . . . 853
23.878 so_17492092_17492530_1 . . . 853
23.879 so_17492092_17492530_0 . . . 853
23.880 so_28136080_28139567_2 . . . 853
23.881 so_22540292_22540625_0 . . . 853
23.882 so_24708480_24708554_1 . . . 853
23.883 so_26718829_26719784_0 . . . 853
23.884 so_1155693_1155721_3 . . . 853
23.885 so_28277982_28277982_4 . . . 853
23.886 so_28277982_30244945_4 . . . 853
23.887 so_28266382_28266883_2 . . . 853
23.888 so_28266382_28266883_1 . . . 853
23.889 so_29899282_29899835_2 . . . 853
23.890 so_12982668_12982926_0 . . . 853
23.891 so_12433154_12433681_0 . . . 853
23.892 so_39055936_39056102_0 . . . 855
23.893 so_15126881_15126913_3 . . . 855
23.894 Associative containers . . . 856
23.895 so_36475497_36478025_1 . . . 858
23.896 so_36475497_36475910_0 . . . 858
23.897 so_16382948_16382969_1 . . . 861
23.898 so_31347198_31347486_1 . . . 861
23.899 so_32247436_32247587_0 . . . 861
23.900 so_32247436_32247520_0 . . . 861
23.901 so_29977259_29977450_0 . . . 861
23.902 so_25786276_25786691_11 . . . 861
23.903 so_25786276_25786691_2 . . . 861
23.904 so_35617844_35617887_1 . . . 861
23.905 so_16838591_16838644_1 . . . 861
23.906 so_7627805_7627823_0 . . . 861
23.907 so_8943261_8943287_3 . . . 861
23.908 so_20574035_20574126_0 . . . 861
23.909 so_40098080_40098140_0 . . . 861
23.910 so_36617248_36617442_2 . . . 861
23.911 so_4523959_12965621_1 . . . 861
23.912 so_21682716_21682858_0 . . . 861
23.913 so_31347198_31347486_2 . . . 861
23.914 so_25786276_25786691_3 . . . 861
23.915 so_25786276_25786691_12 . . . 861
23.916 so_16382948_16382969_3 . . . 862
23.917 so_25786276_25786691_7 . . . 862
23.918 so_25786276_25786691_15 . . . 862
23.919 so_36617248_36617442_4 . . . 862
23.920 so_15080742_15080926_2 . . . 862
23.921 so_17312619_17312804_6 . . . 862
23.922 so_46770274_46777886_1 . . . 862
23.923 so_7357331_7357352_1 . . . 862
23.924 so_17312619_17312804_7 . . . 862
23.925 so_17552191_17553165_0 . . . 862

23.926 so_19159753_19160538_1 . . . 862
23.927 so_19159753_19160538_0 . . . 862
23.928 so_5632194_5633372_0 . . . 862
23.929 so_5632079_5632131_0 . . . 866
23.930 so_15761239_15761416_0 . . . 866
23.931 so_34430992_34431070_3 . . . 866
23.932 Unordered associative containers . . . 873
23.933 so_19469082_19469261_0 . . . 875
23.934 so_17036428_17036478_1 . . . 878
23.935 so_36323096_36323132_1 . . . 878
23.936 so_46770274_46777886_0 . . . 878
23.937 so_37729367_39692746_1 . . . 879
23.938 so_31772406_31772406_1 . . . 883
23.939 Container adaptors . . . 890
23.940 so_23086665_23088522_1 . . . 890
23.941 so_15284567_15284582_0 . . . 890
23.942 so_25877436_25877494_0 . . . 893
23.943 so_17906131_17906171_0 . . . 893
23.944 so_35635098_35635395_0 . . . 894
23.945 so_35190585_35190714_0 . . . 894
23.946 so_46589317_46589523_0 . . . 895
23.947 so_46589317_46589523_1 . . . 895

24 Iterators library **900**
24.1 General . . . 900
24.2 Iterator requirements . . . 900
24.3 so_23567244_23567485_1 . . . 900
24.4 so_28288110_28288167_6 . . . 900
24.5 so_21826187_21828507_0 . . . 900
24.6 so_35680976_35747288_5 . . . 900
24.7 so_25942025_25942070_0 . . . 900
24.8 so_10364050_10364130_0 . . . 900
24.9 so_5606973_5607012_1 . . . 900
24.10 so_28288110_28288167_7 . . . 900
24.11 so_35680976_35747288_4 . . . 900
24.12 so_25942025_25942070_2 . . . 900
24.13 so_41853089_41853537_1 . . . 900
24.14 so_28288110_28288167_8 . . . 900
24.15 so_14058642_14058712_1 . . . 900
24.16 so_12608360_12608431_0 . . . 900
24.17 so_13266723_13266830_1 . . . 900
24.18 so_11929020_11929020_0 . . . 901
24.19 so_40901372_40901430_0 . . . 901
24.20 so_12608360_12608431_1 . . . 901
24.21 so_13266723_13266830_2 . . . 901
24.22 so_1190112_1190275_0 . . . 901
24.23 so_5441893_5442120_2 . . . 901
24.24 so_5441893_5441944_1 . . . 901
24.25 so_5441893_5441893_0 . . . 901
24.26 so_5441893_5441944_0 . . . 901
24.27 so_4285895_20614158_0 . . . 901

24.28 so_32193096_32193172_3 . . . 901
24.29 so_30603512_30603512_0 . . . 901
24.30 so_15058414_15058433_1 . . . 901
24.31 so_21053273_21213706_1 . . . 901
24.32 so_8214116_8214201_0 . . . 901
24.33 so_8214116_8214256_0 . . . 901
24.34 so_26241878_26241878_0 . . . 901
24.35 so_27069432_27069592_0 . . . 901
24.36 so_33657050_33675281_11 . . . 901
24.37 so_35711301_35711873_1 . . . 901
24.38 so_16267615_16267741_1 . . . 901
24.39 so_15692998_15693393_0 . . . 901
24.40 so_23680181_39521025_3 . . . 901
24.41 so_30714156_30714156_0 . . . 901
24.42 so_19744771_19744858_0 . . . 901
24.43 so_32748870_32789107_0 . . . 901
24.44 so_2362454_2363369_0 . . . 901
24.45 so_5113365_5113365_0 . . . 901
24.46 so_20725030_20730923_1 . . . 901
24.47 so_8396484_8396581_0 . . . 901
24.48 so_31792101_31792573_0 . . . 901
24.49 so_3395180_3395263_0 . . . 901
24.50 so_43501525_43501589_1 . . . 901
24.51 so_6085531_19257003_0 . . . 901
24.52 so_33076032_33076132_0 . . . 901
24.53 so_17198239_17198612_0 . . . 901
24.54 so_14592215_14592233_1 . . . 901
24.55 so_19480609_19483347_2 . . . 901
24.56 so_16100865_16101066_0 . . . 901
24.57 so_36007044_36007884_1 . . . 901
24.58 so_7505267_7505576_0 . . . 901
24.59 so_7756747_7756766_0 . . . 901
24.60 so_39309908_39309980_3 . . . 901
24.61 so_39309908_39309980_1 . . . 901
24.62 so_4657513_4664519_0 . . . 901
24.63 so_29654728_29654925_0 . . . 901
24.64 so_7504921_7505121_0 . . . 901
24.65 so_7504921_7505034_2 . . . 901
24.66 so_12520471_12520690_0 . . . 901
24.67 so_19480609_19483347_1 . . . 901
24.68 so_17492160_17492236_1 . . . 901
24.69 so_15142603_15142704_2 . . . 901
24.70 so_7505267_7505576_2 . . . 901
24.71 so_24290711_24290895_0 . . . 901
24.72 so_16267615_16267675_1 . . . 901
24.73 so_7756747_7756766_1 . . . 901
24.74 so_39309908_39309980_4 . . . 901
24.75 so_39309908_39309980_2 . . . 901
24.76 so_25629061_25630943_1 . . . 901
24.77 so_23761273_23761589_3 . . . 901
24.78 so_23761273_23761539_0 . . . 901

24.79 so_7504921_7505121_2 . . . 901
24.80 so_7504921_7505034_3 . . . 901
24.81 so_7504921_7505070_1 . . . 901
24.82 so_19353156_19355116_0 . . . 901
24.83 so_13382622_13382801_0 . . . 901
24.84 so_11779859_11780212_0 . . . 901
24.85 so_43387247_43387721_0 . . . 901
24.86 so_43268997_43270195_0 . . . 901
24.87 so_4285895_20614158_2 . . . 901
24.88 so_21053273_21213706_3 . . . 901
24.89 so_28684573_28687991_0 . . . 901
24.90 so_20725030_20730923_3 . . . 901
24.91 so_43501525_43501589_0 . . . 901
24.92 so_33076032_33076132_1 . . . 901
24.93 so_23567244_23567485_0 . . . 901
24.94 so_38468630_38469498_3 . . . 901
24.95 so_28631332_28631860_0 . . . 901
24.96 so_17992821_17992917_2 . . . 901
24.97 so_35680976_35681139_0 . . . 902
24.98 so_35680976_35681139_1 . . . 902
24.99 so_35680976_35747288_2 . . . 902
24.100 so_35243358_35245058_0 . . . 902
24.101 so_16900498_16901637_0 . . . 902
24.102 so_5947683_5951860_4 . . . 902
24.103 so_22038056_22038056_0 . . . 902
24.104 so_11887104_11938988_0 . . . 902
24.105 so_11887104_11887104_0 . . . 902
24.106 so_33657050_33675281_13 . . . 902
24.107 so_23301162_23301404_5 . . . 902
24.108 so_8869104_8869179_0 . . . 902
24.109 so_35243358_35243358_1 . . . 902
24.110 so_35243358_35243358_0 . . . 902
24.111 so_5947683_5947683_0 . . . 903
24.112 so_5947683_5947683_1 . . . 903
24.113 so_15924799_15924799_0 . . . 903
24.114 so_8869104_8869179_1 . . . 903
24.115 so_4235165_4235360_0 . . . 903
24.116 so_17663449_17663654_1 . . . 903
24.117 so_19480609_19481123_1 . . . 903
24.118 so_2537130_2537183_1 . . . 903
24.119 so_35141436_35141554_1 . . . 904
24.120 so_3049597_3049609_0 . . . 904
24.121 so_11876128_11876128_1 . . . 904
24.122 so_11876128_11876128_0 . . . 904
24.123 so_13870601_13870601_1 . . . 904
24.124 so_13870601_13870601_2 . . . 904
24.125 so_14058642_14058712_0 . . . 904
24.126 so_30417500_30417581_0 . . . 904
24.127 so_46505977_46506185_1 . . . 904
24.128 so_13867539_13867678_0 . . . 904
24.129 so_11281298_13260745_2 . . . 904

24.130 so_11281298_13260745_0 . . . 904
24.131 so_11281298_13260745_4 . . . 904
24.132 so_11281298_13260745_1 . . . 904
24.133 so_11281298_13260745_3 . . . 904
24.134 so_8869104_8869179_2 . . . 904
24.135 so_10396278_10396955_1 . . . 904
24.136 so_20114819_20114908_0 . . . 904
24.137 so_20114819_20114975_0 . . . 904
24.138 so_24059532_24059774_3 . . . 904
24.139 so_24859092_24859898_1 . . . 904
24.140 so_12608360_12608431_2 . . . 904
24.141 so_28631332_28631860_1 . . . 904
24.142 so_28631332_28631860_2 . . . 904
24.143 so_46626832_46626887_0 . . . 904
24.144 so_28679881_28680126_0 . . . 904
24.145 so_2549957_2549991_1 . . . 904
24.146 so_1190112_24145199_0 . . . 904
24.147 so_16100865_16101066_1 . . . 904
24.148 so_26241878_26241878_1 . . . 904
24.149 so_24859092_24859898_2 . . . 904
24.150 so_4657513_4664519_1 . . . 904
24.151 so_6302706_6303180_0 . . . 904
24.152 so_11929020_11929299_0 . . . 904
24.153 so_11929020_11929299_1 . . . 904
24.154 so_8869104_8869179_3 . . . 904
24.155 so_24859092_24859898_3 . . . 904
24.156 so_11929020_11929299_2 . . . 904
24.157 so_41601734_41602076_0 . . . 905
24.158 so_4657513_4658175_1 . . . 905
24.159 so_13870601_13870601_5 . . . 905
24.160 so_13417236_13417340_0 . . . 905
24.161 so_41898473_41899111_12 . . . 905
24.162 so_41898473_41899111_15 . . . 905
24.163 so_41601734_41602076_1 . . . 905
24.164 so_4657513_4658175_2 . . . 905
24.165 so_35390835_35390932_0 . . . 905
24.166 so_29476666_29476950_0 . . . 905
24.167 so_24538500_24538585_0 . . . 905
24.168 so_33843939_33844489_0 . . . 905
24.169 so_36946856_36946928_1 . . . 905
24.170 so_29056636_29056709_1 . . . 905
24.171 Header `<iterator>` synopsis . . . 906
24.172 Iterator primitives . . . 909
24.173 so_44019135_44061435_1 . . . 909
24.174 so_44019135_44061435_2 . . . 909
24.175 so_44019135_44061435_3 . . . 909
24.176 so_21826187_21828507_2 . . . 909
24.177 so_16874183_16905832_0 . . . 909
24.178 so_4335962_31409532_1 . . . 909
24.179 so_4335962_31409532_0 . . . 909
24.180 so_1107149_1107235_1 . . . 909

24.181 so_1107149_1107235_0 . . . 909
24.182 so_33805613_33805810_1 . . . 909
24.183 so_10484015_10485296_0 . . . 911
24.184 so_47513930_47515462_0 . . . 911
24.185 so_46476150_46476150_0 . . . 911
24.186 so_5916239_5916501_0 . . . 912
24.187 so_16663541_16664361_0 . . . 912
24.188 so_3087364_3087378_0 . . . 912
24.189 so_14369499_14370750_2 . . . 912
24.190 so_31907580_31907715_3 . . . 912
24.191 so_1668088_1668111_0 . . . 912
24.192 so_43387247_43387721_1 . . . 912
24.193 so_38305559_38306069_0 . . . 912
24.194 so_38305559_38306069_1 . . . 912
24.195 so_31907580_31907715_1 . . . 913
24.196 so_29654728_29654728_0 . . . 913
24.197 Iterator adaptors . . . 913
24.198 so_10085570_10086036_0 . . . 913
24.199 so_10261705_10262413_0 . . . 913
24.200 so_33881285_33881710_1 . . . 913
24.201 so_25263676_25264552_0 . . . 913
24.202 so_10373764_10373903_0 . . . 913
24.203 so_24997910_24998000_0 . . . 913
24.204 so_2537130_2537183_0 . . . 918
24.205 so_44767368_44767477_0 . . . 918
24.206 so_28199082_28199654_0 . . . 922
24.207 Stream iterators . . . 926
24.208 so_29495546_29496571_0 . . . 926
24.209 so_17662413_17662966_0 . . . 926
24.210 so_5074122_5074122_0 . . . 926
24.211 so_26241878_26241969_0 . . . 926
24.212 so_17776023_17776162_0 . . . 926
24.213 so_16370924_16371110_0 . . . 926
24.214 so_26241878_26241969_1 . . . 926
24.215 so_1702929_1703247_1 . . . 926
24.216 so_35575687_35577045_1 . . . 927
24.217 so_5074122_5074122_1 . . . 927
24.218 so_12602657_12602770_0 . . . 927
24.219 so_28366444_28368013_8 . . . 928
24.220 so_46955960_46956700_0 . . . 931
24.221 range access . . . 933
24.222 so_6882561_6882643_0 . . . 933
24.223 so_45659489_45684064_0 . . . 933
24.224 so_38878454_38878818_0 . . . 933

25 Algorithms library **935**
25.1 General . . . 935
25.2 so_32721635_32721797_0 . . . 945
25.3 so_28033901_28033990_1 . . . 945
25.4 so_19480609_19481123_0 . . . 945
25.5 so_29056636_29056709_0 . . . 945

25.6 so_32721635_32721797_1 . . . 945
25.7 so_27764277_27767611_0 . . . 945
25.8 so_22544528_22544697_2 . . . 945
25.9 so_22544528_22544697_1 . . . 945
25.10 so_26245969_26246172_0 . . . 946
25.11 so_24072180_24072279_0 . . . 946
25.12 so_47353796_47354427_1 . . . 946
25.13 so_41103743_41104716_0 . . . 946
25.14 so_34825552_34829349_1 . . . 946
25.15 so_4885676_4885823_0 . . . 946
25.16 so_41108386_41108386_0 . . . 946
25.17 so_25013168_25013874_1 . . . 946
25.18 so_24117997_24118439_2 . . . 946
25.19 so_24117997_24118439_1 . . . 946
25.20 so_17626667_17627391_0 . . . 946
25.21 so_34522254_34522287_0 . . . 946
25.22 so_23301162_23301404_0 . . . 946
25.23 Non-modifying sequence operations . . . 946
25.24 so_25600484_25600513_0 . . . 946
25.25 so_18418470_18418841_0 . . . 947
25.26 so_21004879_21005208_2 . . . 947
25.27 so_29046016_29046016_0 . . . 947
25.28 so_35277564_35278527_0 . . . 947
25.29 so_14838672_14838672_0 . . . 947
25.30 so_21004879_21005208_3 . . . 947
25.31 so_7504921_7505070_0 . . . 947
25.32 so_35277564_35278527_1 . . . 947
25.33 so_35277564_35278267_1 . . . 947
25.34 so_21004879_21005208_5 . . . 947
25.35 so_7504921_7505091_0 . . . 947
25.36 so_21004879_21005208_6 . . . 947
25.37 so_19216123_19216133_0 . . . 947
25.38 so_36816245_36816463_0 . . . 947
25.39 so_17231829_17232247_1 . . . 947
25.40 so_31946599_31946802_0 . . . 947
25.41 so_14369499_14370750_3 . . . 947
25.42 so_26187259_26187375_3 . . . 947
25.43 so_26187259_26187375_1 . . . 948
25.44 so_9155476_9155486_0 . . . 948
25.45 so_17901678_17903794_4 . . . 949
25.46 so_21680251_21680416_0 . . . 949
25.47 so_19120779_19120926_0 . . . 950
25.48 so_21680251_21680416_1 . . . 950
25.49 so_21680251_21680416_2 . . . 950
25.50 so_15209006_15209067_0 . . . 950
25.51 so_4940432_4940677_1 . . . 951
25.52 so_26187259_26187375_0 . . . 951
25.53 so_26187259_26187375_2 . . . 951
25.54 Mutating sequence operations . . . 952
25.55 so_19480609_19483347_3 . . . 952
25.56 so_19480609_19483347_0 . . . 952

25.57 so_4484900_4484916_0 . . . 952
25.58 so_6230180_6230493_1 . . . 952
25.59 so_23301162_23301404_2 . . . 952
25.60 so_29016785_29017032_1 . . . 952
25.61 so_1840121_1840182_1 . . . 952
25.62 so_17663449_17663654_0 . . . 952
25.63 so_19480609_19483347_6 . . . 952
25.64 so_29016785_29017032_4 . . . 952
25.65 so_42720353_42721669_3 . . . 952
25.66 so_17356719_33306492_1 . . . 954
25.67 so_19200528_19200789_0 . . . 954
25.68 so_17356719_33306492_2 . . . 954
25.69 so_17356719_33306492_4 . . . 954
25.70 so_19200528_19200789_2 . . . 954
25.71 so_37815377_37815620_1 . . . 954
25.72 so_37815377_37815620_0 . . . 954
25.73 so_17356719_33306492_5 . . . 954
25.74 so_33283275_33305627_1 . . . 955
25.75 so_33283275_33283491_0 . . . 955
25.76 so_9380190_9380451_0 . . . 955
25.77 so_14823732_14823732_0 . . . 955
25.78 so_33283275_33305627_2 . . . 955
25.79 so_33283275_33305627_3 . . . 955
25.80 so_33283275_33305627_4 . . . 956
25.81 so_9380190_9380451_1 . . . 956
25.82 so_17901678_17903794_0 . . . 956
25.83 so_28199082_28199654_1 . . . 956
25.84 so_8415016_8415207_0 . . . 956
25.85 so_17901678_17903794_1 . . . 956
25.86 so_37281927_37281927_0 . . . 956
25.87 so_25419851_25420087_0 . . . 956
25.88 so_17901678_17903794_5 . . . 956
25.89 so_8415016_8415207_1 . . . 956
25.90 so_44759353_44760075_1 . . . 956
25.91 so_24052415_31929468_0 . . . 957
25.92 so_38005753_38005874_0 . . . 957
25.93 so_38005753_38005753_1 . . . 957
25.94 so_37582510_37582863_1 . . . 957
25.95 so_31586277_31591251_0 . . . 957
25.96 so_4972470_4972489_0 . . . 958
25.97 so_1900924_1900938_0 . . . 959
25.98 Sorting and related operations . . . 960
25.99 so_8013599_8014673_0 . . . 960
25.100 so_18110051_18110127_0 . . . 960
25.101 so_17906131_17906171_1 . . . 960
25.102 so_36007500_36007500_0 . . . 960
25.103 so_8013599_8014673_1 . . . 960
25.104 so_17631674_17631885_0 . . . 960
25.105 so_23927003_23927135_0 . . . 960
25.106 so_18110051_18110127_1 . . . 960
25.107 so_31825064_31825767_0 . . . 960

25.108 so_10776931_10777144_0 . . . 960
25.109 so_18958015_18958250_0 . . . 960
25.110 so_32246949_32247189_0 . . . 960
25.111 so_45905122_45905608_0 . . . 960
25.112 so_22461779_22462260_0 . . . 960
25.113 so_38071240_38072162_0 . . . 960
25.114 so_17631674_17631885_1 . . . 960
25.115 so_23927003_23927135_1 . . . 960
25.116 so_18958015_18958178_0 . . . 960
25.117 so_18291620_18292045_1 . . . 960
25.118 so_44881979_44883041_6 . . . 960
25.119 so_24286209_24286209_0 . . . 960
25.120 so_35287049_35287049_1 . . . 960
25.121 so_32246949_32247189_1 . . . 960
25.122 so_46863507_46863716_0 . . . 960
25.123 so_45905122_45905608_1 . . . 960
25.124 so_35502809_35502911_0 . . . 960
25.125 so_34834125_34834389_0 . . . 960
25.126 so_34834125_34834157_0 . . . 960
25.127 so_18525398_18525472_0 . . . 960
25.128 so_38071240_38072162_1 . . . 960
25.129 so_38071240_38072162_3 . . . 960
25.130 so_38071240_38072162_2 . . . 960
25.131 so_12419603_12419851_4 . . . 960
25.132 so_12419603_12419851_1 . . . 960
25.133 so_12419603_12419851_5 . . . 960
25.134 so_12419603_12419851_0 . . . 960
25.135 so_12419603_12419851_2 . . . 960
25.136 so_18291620_18292045_2 . . . 960
25.137 so_32246949_32247189_2 . . . 960
25.138 so_46863507_46863716_1 . . . 960
25.139 so_3638406_3641623_0 . . . 960
25.140 so_41180181_41180281_1 . . . 960
25.141 so_8096817_8097097_0 . . . 960
25.142 so_34834125_34834157_1 . . . 960
25.143 so_18525398_18525472_1 . . . 960
25.144 so_18525398_18525472_2 . . . 960
25.145 so_8219525_8219718_0 . . . 961
25.146 so_18110051_18110051_0 . . . 961
25.147 so_47113457_47113740_0 . . . 961
25.148 so_32966824_32969444_4 . . . 961
25.149 so_9539014_9539344_1 . . . 961
25.150 so_45455345_45457821_3 . . . 961
25.151 so_4484900_24481485_1 . . . 961
25.152 so_30883069_30883224_4 . . . 961
25.153 so_7505267_7505267_0 . . . 961
25.154 so_32966824_32969444_5 . . . 961
25.155 so_4484900_24481485_3 . . . 961
25.156 so_44947510_44948166_0 . . . 961
25.157 so_45455345_45457821_2 . . . 962
25.158 so_45455345_45457821_4 . . . 962

25.159 so_18384269_18384600_0 . . . 963
25.160 so_11245359_11245427_0 . . . 963
25.161 so_13502128_13502213_1 . . . 963
25.162 so_44881979_44883041_8 . . . 963
25.163 so_32034115_32034260_1 . . . 963
25.164 so_15121477_15121738_0 . . . 963
25.165 so_44881979_44883041_1 . . . 963
25.166 so_18767829_18767996_0 . . . 963
25.167 so_13480679_13481498_0 . . . 963
25.168 so_38980284_38980347_0 . . . 963
25.169 so_5072257_5072289_0 . . . 964
25.170 so_6768885_6768885_1 . . . 964
25.171 so_15121477_15121738_1 . . . 964
25.172 so_44881979_44883041_2 . . . 964
25.173 so_18767829_18767996_1 . . . 964
25.174 so_15121477_15121738_3 . . . 964
25.175 so_15121477_15121738_4 . . . 964
25.176 so_18767829_18767996_3 . . . 964
25.177 so_9674136_9674190_0 . . . 964
25.178 so_32276196_32276421_0 . . . 964
25.179 so_18767829_18767996_4 . . . 964
25.180 so_30495744_30495744_0 . . . 965
25.181 so_17603507_17603578_0 . . . 965
25.182 so_30495744_30495744_1 . . . 965
25.183 so_21191631_21191731_0 . . . 965
25.184 so_21624268_21624819_3 . . . 965
25.185 so_41992207_41992244_0 . . . 965
25.186 so_21624268_21624819_0 . . . 965
25.187 so_25745759_25758367_12 . . . 965
25.188 so_45455345_45457821_1 . . . 965
25.189 so_21624268_21624819_1 . . . 966
25.190 so_26632809_26633267_0 . . . 966
25.191 so_5818687_5819015_0 . . . 966
25.192 so_43604746_43604876_3 . . . 966
25.193 so_3195973_3196065_0 . . . 966
25.194 so_3195973_3196071_0 . . . 966
25.195 so_8596514_8596656_1 . . . 967
25.196 so_42285608_42285659_0 . . . 968
25.197 so_2974470_2974501_1 . . . 969
25.198 so_2974470_2974501_0 . . . 969
25.199 so_17890219_17890265_1 . . . 969
25.200 so_32966824_32969444_1 . . . 970
25.201 so_32966824_32969444_2 . . . 970
25.202 so_17088900_17088983_1 . . . 970
25.203 so_17088900_17088983_2 . . . 971
25.204 so_33175026_33175718_0 . . . 972
25.205 C library algorithms . . . 973
25.206 so_26160465_26160651_3 . . . 973
25.207 so_1571340_43877628_1 . . . 973
25.208 so_26160962_26160962_0 . . . 973
25.209 so_26160465_26160651_6 . . . 974

25.210 so_26160962_26160962_3 . . . 974
25.211 so_8380805_8380835_2 . . . 974

26 Numerics library **975**
26.1 General . . . 975
26.2 Numeric type requirements . . . 975
26.3 so_2187648_2187963_0 . . . 976
26.4 so_2187648_2187963_1 . . . 976
26.5 The floating-point environment . . . 976
26.6 so_16188263_16188263_1 . . . 977
26.7 so_44694574_44694661_1 . . . 977
26.8 so_36218367_36219717_5 . . . 977
26.9 so_39205308_39206454_1 . . . 977
26.10 so_39205308_39206454_0 . . . 977
26.11 Complex numbers . . . 977
26.12 so_44180119_44180119_1 . . . 977
26.13 so_11108587_11108653_0 . . . 977
26.14 so_11108587_11108638_0 . . . 977
26.15 so_44180119_44180119_2 . . . 977
26.16 so_25348099_25348164_0 . . . 977
26.17 so_23519366_23520299_0 . . . 977
26.18 so_9476036_9476111_0 . . . 977
26.19 so_44180119_44180119_3 . . . 977
26.20 so_20862774_20862774_10 . . . 977
26.21 so_23198943_23198943_0 . . . 977
26.22 so_30245014_30245429_0 . . . 977
26.23 so_14569690_14569958_1 . . . 977
26.24 so_5020076_5020268_2 . . . 977
26.25 so_5020076_5020268_1 . . . 977
26.26 so_5020076_5020268_0 . . . 977
26.27 so_19175776_19179387_0 . . . 977
26.28 so_44180119_44180119_4 . . . 977
26.29 so_44180119_44180119_7 . . . 977
26.30 so_44180119_44180119_6 . . . 977
26.31 so_22925905_22925905_0 . . . 977
26.32 so_25348099_25348164_3 . . . 977
26.33 so_25348099_25348164_2 . . . 977
26.34 so_23519366_23520299_1 . . . 983
26.35 so_21081275_21081323_0 . . . 984
26.36 so_16730716_16731034_0 . . . 986
26.37 so_15780011_15780493_0 . . . 986
26.38 so_17264811_17264847_1 . . . 987
26.39 so_8478338_8478591_2 . . . 987
26.40 so_8478338_8478591_1 . . . 987
26.41 so_8478338_8478591_0 . . . 987
26.42 so_8478338_8478591_3 . . . 987
26.43 so_41688952_41688952_1 . . . 987
26.44 so_31800065_31800404_0 . . . 987
26.45 Random number generation . . . 987
26.46 so_27962421_27963240_0 . . . 987
26.47 so_11767169_11767169_0 . . . 988

26.48 so_23690099_23690146_1 . . . 988
26.49 so_23690099_23690146_2 . . . 988
26.50 so_23690099_23690146_0 . . . 988
26.51 so_41688952_41688952_4 . . . 988
26.52 so_31460733_31460827_2 . . . 988
26.53 so_31460733_31460733_0 . . . 988
26.54 so_31460733_31460827_1 . . . 988
26.55 so_5939486_5939961_0 . . . 988
26.56 so_30035547_30035720_0 . . . 988
26.57 so_20808905_20809251_0 . . . 988
26.58 so_39591067_39591067_1 . . . 988
26.59 so_45801696_45801696_0 . . . 988
26.60 so_21096908_21097439_0 . . . 988
26.61 so_41777551_41777846_0 . . . 988
26.62 so_41777551_41782071_0 . . . 988
26.63 so_41777551_41782071_2 . . . 988
26.64 so_25298585_25298621_2 . . . 988
26.65 so_25298585_25298621_0 . . . 988
26.66 so_38269978_38269978_0 . . . 988
26.67 so_38269978_38269978_2 . . . 988
26.68 so_38269978_38269978_1 . . . 988
26.69 so_31197713_31197713_2 . . . 988
26.70 so_31197713_31197713_0 . . . 988
26.71 so_22522829_22523086_1 . . . 989
26.72 so_22522829_22523086_0 . . . 989
26.73 so_35680976_35747288_3 . . . 989
26.74 so_35680976_35681139_2 . . . 989
26.75 so_13268608_13268608_0 . . . 989
26.76 so_35994505_35995627_1 . . . 990
26.77 so_25129500_25130000_1 . . . 990
26.78 so_24311003_24311199_0 . . . 990
26.79 so_19178647_19178785_3 . . . 990
26.80 so_25129500_25130000_3 . . . 990
26.81 so_19178647_19178785_4 . . . 990
26.82 so_29434001_29434142_3 . . . 991
26.83 so_17908425_17912424_0 . . . 991
26.84 so_18299631_18299631_0 . . . 991
26.85 so_25129500_25130000_2 . . . 991
26.86 so_37380430_37380652_1 . . . 991
26.87 so_19178647_19178785_0 . . . 991
26.88 so_34365746_34365963_1 . . . 991
26.89 so_35233468_35234495_2 . . . 994
26.90 so_26714209_26714209_0 . . . 996
26.91 so_16349006_16351405_0 . . . 997
26.92 so_24566574_24566792_0 . . . 997
26.93 so_34192532_34192994_0 . . . 997
26.94 so_19036141_19036382_0 . . . 997
26.95 so_26947324_26947349_0 . . . 997
26.96 so_26714209_26715518_0 . . . 997
26.97 so_28328948_28329163_0 . . . 997
26.98 so_30196790_30197397_0 . . . 999

26.99 so_24105805_24128288_0 . . . 1001
26.100 so_37927205_37927205_0 . . . 1001
26.101 so_10896915_10896934_0 . . . 1001
26.102 so_28607889_28610763_0 . . . 1008
26.103 so_26961789_26961867_0 . . . 1008
26.104 so_37945263_37949721_1 . . . 1008
26.105 so_36553935_36553990_1 . . . 1008
26.106 so_28390843_29575664_0 . . . 1009
26.107 so_42685506_42685720_1 . . . 1009
26.108 so_28390843_29575664_1 . . . 1009
26.109 so_45069219_45069347_1 . . . 1009
26.110 so_32275328_32275420_0 . . . 1009
26.111 so_9549357_9574589_2 . . . 1009
26.112 so_20591781_20591943_1 . . . 1012
26.113 so_25668600_25668899_0 . . . 1012
26.114 so_36957670_36957670_0 . . . 1012
26.115 so_38532927_38533088_0 . . . 1012
26.116 so_38532927_38533067_5 . . . 1012
26.117 so_28607889_28610763_1 . . . 1012
26.118 so_32746092_32746283_0 . . . 1012
26.119 so_47289571_47290014_0 . . . 1012
26.120 so_32746092_32746283_1 . . . 1012
26.121 so_34787758_34787826_0 . . . 1013
26.122 so_24477218_24477911_0 . . . 1013
26.123 so_38532927_38533067_3 . . . 1022
26.124 so_38532927_38533067_1 . . . 1022
26.125 so_34485220_34485250_2 . . . 1022
26.126 so_34485220_34485250_0 . . . 1022
26.127 so_27492338_27492474_2 . . . 1022
26.128 so_27492338_27492474_0 . . . 1022
26.129 Numeric arrays . . . 1032
26.130 so_9585990_9588075_4 . . . 1034
26.131 so_1602451_10640258_0 . . . 1034
26.132 so_1353757_1353813_0 . . . 1035
26.133 so_11946711_11946896_1 . . . 1035
26.134 so_11946711_11946896_2 . . . 1035
26.135 so_2187648_2187963_3 . . . 1037
26.136 so_12580628_12580789_1 . . . 1037
26.137 so_31619467_31619467_1 . . . 1037
26.138 so_31866735_31867026_2 . . . 1038
26.139 so_31866735_31867026_6 . . . 1038
26.140 so_11143109_11143184_0 . . . 1039
26.141 so_2187648_2187963_5 . . . 1039
26.142 so_1353757_1353782_0 . . . 1045
26.143 so_12608360_12608599_0 . . . 1053
26.144 Generalized numeric operations . . . 1053
26.145 so_28862947_28863128_0 . . . 1054
26.146 so_16854768_16854951_1 . . . 1054
26.147 so_26805502_26805636_2 . . . 1055
26.148 so_46135838_46142707_0 . . . 1055
26.149 C library . . . 1056

26.150 so_12696764_12696998_0 . . . 1056
26.151 so_3297201_3297247_0 . . . 1058
26.152 so_37945263_37949721_0 . . . 1058
26.153 so_13460750_13460915_0 . . . 1058
26.154 so_19613191_19684955_0 . . . 1058
26.155 so_37952267_37952576_0 . . . 1058
26.156 so_12649478_12649494_0 . . . 1058
26.157 so_42959061_42959220_0 . . . 1058
26.158 so_13460750_13460915_3 . . . 1058
26.159 so_44369648_44370539_5 . . . 1060
26.160 so_38798791_38799236_1 . . . 1060
26.161 so_27336479_27336856_0 . . . 1060
26.162 so_11826314_11826361_0 . . . 1060
26.163 so_12063821_12063878_0 . . . 1061
26.164 so_30084577_30084734_0 . . . 1061
26.165 so_30084577_30084734_1 . . . 1061
26.166 so_30766491_30766492_0 . . . 1061
26.167 so_30766491_30766492_3 . . . 1061
26.168 so_5627030_5627278_3 . . . 1061
26.169 so_5627030_5627278_2 . . . 1061
26.170 so_5627030_5627278_1 . . . 1061
26.171 so_5627030_5627278_0 . . . 1061
26.172 so_19613191_19684955_1 . . . 1061
26.173 so_19613191_19684955_2 . . . 1061
26.174 so_19249975_19250182_0 . . . 1061
26.175 so_19249975_19250182_1 . . . 1061
26.176 so_17264811_17264847_0 . . . 1061
26.177 so_15572705_15572705_0 . . . 1061
26.178 so_15572705_15573448_0 . . . 1061
26.179 so_14295217_14298922_0 . . . 1061
26.180 so_27336479_27336856_1 . . . 1061
26.181 so_40461330_40461330_2 . . . 1061
26.182 so_40461330_40461330_0 . . . 1061
26.183 so_40461330_40461330_1 . . . 1061
26.184 so_40461330_40461330_3 . . . 1061
26.185 so_42959061_42959220_1 . . . 1061
26.186 so_42959061_42959220_2 . . . 1061
26.187 so_42959061_42959220_4 . . . 1061
26.188 so_42959061_42959220_3 . . . 1061
26.189 so_29750946_29750947_1 . . . 1061
26.190 so_29750946_29750947_0 . . . 1061

27 Input/output library **1062**
27.1 General . . . 1062
27.2 Iostreams requirements . . . 1063
27.3 so_26701268_26701456_0 . . . 1063
27.4 so_28294129_28975080_0 . . . 1063
27.5 so_28294129_28975080_2 . . . 1063
27.6 so_25126803_25127119_0 . . . 1063
27.7 so_29870879_29871321_0 . . . 1063
27.8 so_22927197_22931994_0 . . . 1063

27.9 so_15513549_15513639_0 . . . 1063
27.10 so_18870549_18870842_0 . . . 1063
27.11 so_42177791_42177900_0 . . . 1063
27.12 so_20211935_20212155_1 . . . 1063
27.13 so_29870879_29871321_1 . . . 1063
27.14 so_22927197_22931994_1 . . . 1063
27.15 so_18870549_18870842_1 . . . 1063
27.16 Forward declarations . . . 1063
27.17 Standard iostream objects . . . 1065
27.18 so_47365800_47365850_0 . . . 1066
27.19 so_2262232_2262332_1 . . . 1066
27.20 so_26975876_26976258_0 . . . 1066
27.21 so_9652374_9652432_1 . . . 1066
27.22 so_46691126_46691217_0 . . . 1066
27.23 so_46691126_46691217_1 . . . 1066
27.24 so_46691126_46691217_2 . . . 1066
27.25 so_37706549_37706791_0 . . . 1066
27.26 so_44613079_44613183_2 . . . 1066
27.27 so_44613079_44613293_1 . . . 1066
27.28 so_27059837_27059902_0 . . . 1066
27.29 so_18711866_18712002_1 . . . 1066
27.30 so_8784892_8784923_0 . . . 1066
27.31 so_40159890_40160218_0 . . . 1066
27.32 so_40159890_40159890_0 . . . 1066
27.33 so_40159890_40160218_1 . . . 1066
27.34 so_20779400_20779486_1 . . . 1066
27.35 so_4298027_4298232_0 . . . 1066
27.36 so_30956795_30956942_2 . . . 1066
27.37 so_30956795_30956942_1 . . . 1066
27.38 so_8947949_8947949_0 . . . 1066
27.39 so_22950412_22950466_0 . . . 1066
27.40 so_26410497_26410990_0 . . . 1066
27.41 so_39076641_39076977_0 . . . 1066
27.42 so_39092042_39092171_0 . . . 1066
27.43 so_12605725_12608911_0 . . . 1066
27.44 so_27762691_27762766_0 . . . 1066
27.45 so_35538220_35538308_0 . . . 1066
27.46 so_18870549_18870842_2 . . . 1066
27.47 so_42177791_42177900_1 . . . 1066
27.48 so_6374264_6374525_0 . . . 1066
27.49 so_5257509_5257664_0 . . . 1066
27.50 so_2262232_2262332_5 . . . 1066
27.51 so_12130000_12130000_2 . . . 1066
27.52 so_26975876_26976002_0 . . . 1067
27.53 so_30888851_30891329_4 . . . 1067
27.54 so_12130000_12130000_4 . . . 1067
27.55 so_26387054_26520333_0 . . . 1067
27.56 Iostreams base classes . . . 1067
27.57 so_24482028_24482028_0 . . . 1069
27.58 so_24482028_24482042_0 . . . 1069
27.59 so_16888145_16888979_5 . . . 1071

27.60 so_39101598_39101598_0 . . . 1071
27.61 so_7624652_7624720_1 . . . 1071
27.62 so_45647977_45648286_0 . . . 1072
27.63 so_16814969_16815635_0 . . . 1072
27.64 so_26228985_26229035_2 . . . 1072
27.65 so_26228985_26229035_0 . . . 1072
27.66 so_26228985_26229035_1 . . . 1072
27.67 so_10359702_10359723_0 . . . 1073
27.68 so_10359702_10359723_1 . . . 1073
27.69 so_3297831_3297831_0 . . . 1073
27.70 so_15727869_15727869_4 . . . 1073
27.71 so_40528228_40528228_1 . . . 1073
27.72 so_35611069_35611635_0 . . . 1074
27.73 so_13196528_13197252_2 . . . 1074
27.74 so_8062344_8062344_0 . . . 1074
27.75 so_24564434_24567920_1 . . . 1075
27.76 so_5750884_5751016_1 . . . 1075
27.77 so_26387054_26520333_1 . . . 1075
27.78 so_42196926_42212104_2 . . . 1076
27.79 so_17022096_17022451_0 . . . 1079
27.80 so_17022096_17022451_1 . . . 1079
27.81 so_25690636_25690637_4 . . . 1079
27.82 so_6240950_6240980_2 . . . 1079
27.83 so_6240950_6240980_1 . . . 1079
27.84 so_8147850_8147850_2 . . . 1082
27.85 so_20371956_20371956_0 . . . 1082
27.86 so_20371956_20371956_1 . . . 1082
27.87 so_20371956_20371956_2 . . . 1082
27.88 so_32794242_32794242_0 . . . 1082
27.89 so_20371956_20371956_4 . . . 1082
27.90 so_9817806_9818136_2 . . . 1083
27.91 so_35019687_35020134_0 . . . 1083
27.92 Stream buffers . . . 1086
27.93 so_4171652_4171828_0 . . . 1086
27.94 so_31688924_31689283_0 . . . 1086
27.95 so_22538568_22539346_0 . . . 1086
27.96 so_8144347_8144474_0 . . . 1089
27.97 so_6152956_6153010_2 . . . 1090
27.98 so_30970385_30970518_2 . . . 1091
27.99 so_12458187_12458187_2 . . . 1091
27.100 so_4349778_4349778_0 . . . 1092
27.101 so_4349778_4350012_0 . . . 1092
27.102 so_12458187_12458187_5 . . . 1093
27.103 so_33145951_33146001_2 . . . 1094
27.104 so_19105657_19105858_4 . . . 1095
27.105 so_19105657_19105858_3 . . . 1095
27.106 so_19105657_19105858_2 . . . 1095
27.107 so_19105657_19105858_5 . . . 1095
27.108 so_19105657_19105858_7 . . . 1095
27.109 Formatting and manipulators . . . 1095
27.110 so_14591203_14591372_1 . . . 1099

27.111 so_14591203_14591372_0 . . . 1099
27.112 so_15658836_15667188_0 . . . 1099
27.113 so_14615671_14615671_1 . . . 1099
27.114 so_15658836_15667188_1 . . . 1099
27.115 so_16468189_16473878_0 . . . 1099
27.116 so_35019687_35020134_4 . . . 1099
27.117 so_25690636_25690637_1 . . . 1099
27.118 so_13423514_23201350_2 . . . 1100
27.119 so_13423514_23201350_1 . . . 1100
27.120 so_14591203_14591203_2 . . . 1100
27.121 so_21567291_21567292_3 . . . 1100
27.122 so_39704088_39704263_1 . . . 1100
27.123 so_13423514_23201350_3 . . . 1100
27.124 so_6552876_6561367_1 . . . 1100
27.125 so_9502810_9502847_2 . . . 1100
27.126 so_34122193_34124651_1 . . . 1100
27.127 so_23236548_23236827_3 . . . 1100
27.128 so_23236548_23236827_1 . . . 1100
27.129 so_16468189_16473878_4 . . . 1101
27.130 so_35019687_35020842_1 . . . 1101
27.131 so_35019687_35020134_2 . . . 1101
27.132 so_35019687_35020842_0 . . . 1101
27.133 so_13378989_13379959_0 . . . 1101
27.134 so_24689378_24689378_1 . . . 1101
27.135 so_13196528_13197252_3 . . . 1101
27.136 so_23374095_23374456_0 . . . 1101
27.137 so_18917167_18918372_0 . . . 1101
27.138 so_18917167_18918372_1 . . . 1101
27.139 so_41622680_41623145_0 . . . 1101
27.140 so_26170513_26170808_0 . . . 1101
27.141 so_35898958_35899150_0 . . . 1101
27.142 so_10818690_10818809_0 . . . 1103
27.143 so_30444641_30444739_0 . . . 1103
27.144 so_10818690_10818809_1 . . . 1103
27.145 so_30444641_30444739_1 . . . 1103
27.146 so_13744897_13745615_0 . . . 1103
27.147 so_4099389_4099576_0 . . . 1103
27.148 so_21809838_21818503_9 . . . 1103
27.149 so_36843616_36847188_1 . . . 1103
27.150 so_30444641_30444739_3 . . . 1103
27.151 so_26984039_26984171_0 . . . 1103
27.152 so_35019687_35020134_1 . . . 1104
27.153 so_31995594_31996600_0 . . . 1104
27.154 so_26642243_26642348_0 . . . 1104
27.155 so_16468189_16473878_3 . . . 1104
27.156 so_13014192_13014761_0 . . . 1104
27.157 so_35088800_35088800_0 . . . 1104
27.158 so_16468189_16473878_2 . . . 1104
27.159 so_24482728_24482728_0 . . . 1104
27.160 so_8688912_8688912_1 . . . 1104
27.161 so_24482728_24482728_2 . . . 1105

27.162 so_34343455_34343563_0 . . . 1105
27.163 so_16814969_16815635_1 . . . 1106
27.164 so_34343455_34343455_0 . . . 1106
27.165 so_6637428_6637643_0 . . . 1107
27.166 so_32298240_32298411_2 . . . 1107
27.167 so_40308080_40308373_0 . . . 1107
27.168 so_2573435_2573572_0 . . . 1107
27.169 so_7855226_7855777_0 . . . 1107
27.170 so_5878052_5878850_0 . . . 1107
27.171 so_13014192_13014542_0 . . . 1107
27.172 so_11712343_11712518_0 . . . 1107
27.173 so_31120621_31127551_0 . . . 1107
27.174 so_26314203_26314321_0 . . . 1107
27.175 so_9191876_9191876_5 . . . 1107
27.176 so_12458187_12458187_10 . . . 1107
27.177 so_10141075_10141203_1 . . . 1108
27.178 so_14819547_14820804_1 . . . 1108
27.179 so_6152956_6153010_1 . . . 1108
27.180 so_526270_526682_0 . . . 1108
27.181 so_6552876_6561367_0 . . . 1108
27.182 so_13732338_13732423_0 . . . 1108
27.183 so_35628288_35628419_1 . . . 1108
27.184 so_25800408_25800622_0 . . . 1109
27.185 so_13423514_23201350_0 . . . 1109
27.186 so_13423514_13423817_0 . . . 1109
27.187 so_11313437_11313437_1 . . . 1110
27.188 so_25690636_25690637_5 . . . 1112
27.189 so_25690636_25690637_2 . . . 1112
27.190 so_6240950_6240980_0 . . . 1112
27.191 so_47438863_47439311_2 . . . 1113
27.192 so_28142886_28143734_2 . . . 1113
27.193 so_28142886_28143734_4 . . . 1113
27.194 so_25690636_25690637_6 . . . 1113
27.195 so_47438863_47439311_3 . . . 1113
27.196 so_6240950_6240980_4 . . . 1113
27.197 so_26516278_26516944_2 . . . 1114
27.198 so_26516278_26516944_4 . . . 1114
27.199 so_32699874_32726664_0 . . . 1114
27.200 so_47438863_47439311_1 . . . 1114
27.201 so_39101598_39132892_0 . . . 1114
27.202 so_37540611_37544789_0 . . . 1114
27.203 so_37112686_37113417_6 . . . 1114
27.204 so_29412717_29412717_2 . . . 1115
27.205 so_30092293_30092348_0 . . . 1115
27.206 so_10053113_10053254_3 . . . 1115
27.207 so_26210379_26210379_0 . . . 1115
27.208 so_16888145_16888979_1 . . . 1115
27.209 so_28746600_28748266_0 . . . 1115
27.210 so_34365746_34365963_8 . . . 1115
27.211 so_20850065_20850142_1 . . . 1116
27.212 so_4633864_4633903_0 . . . 1116

27.213 so_28142886_28143734_0 . . . 1116
27.214 so_11441665_11441904_1 . . . 1116
27.215 so_11441665_11441904_2 . . . 1116
27.216 so_26984039_26984171_1 . . . 1117
27.217 so_11441665_11441904_3 . . . 1117
27.218 so_44380123_45888708_0 . . . 1117
27.219 so_36843616_36847188_3 . . . 1117
27.220 so_43732122_43732803_0 . . . 1117
27.221 so_14526901_14526944_0 . . . 1117
27.222 so_34826388_34826388_0 . . . 1117
27.223 so_14696997_14697203_0 . . . 1117
27.224 so_17736745_17736828_7 . . . 1117
27.225 so_47438863_47439311_0 . . . 1117
27.226 so_25690636_25690637_8 . . . 1118
27.227 so_28142886_28143734_1 . . . 1118
27.228 so_6240950_6240980_3 . . . 1118
27.229 so_1364309_1366028_2 . . . 1118
27.230 so_13014192_13014542_2 . . . 1118
27.231 so_1364309_1366028_3 . . . 1118
27.232 so_30540287_30540674_3 . . . 1119
27.233 so_30540287_30540674_1 . . . 1119
27.234 so_36151077_36151147_0 . . . 1119
27.235 so_34648916_34649021_1 . . . 1119
27.236 so_34648916_34649035_0 . . . 1119
27.237 so_17736745_17736828_1 . . . 1120
27.238 so_17736745_17736828_2 . . . 1120
27.239 so_17736745_17736828_6 . . . 1121
27.240 so_17736745_17736828_5 . . . 1121
27.241 so_13196528_13197252_1 . . . 1121
27.242 so_13196528_13197252_0 . . . 1121
27.243 String-based streams . . . 1124
27.244 so_13169344_13169344_1 . . . 1126
27.245 so_31260386_31260472_2 . . . 1127
27.246 so_20011851_20011912_1 . . . 1127
27.247 so_4349778_4350012_1 . . . 1128
27.248 File-based streams . . . 1135
27.249 so_18187297_18187846_1 . . . 1136
27.250 so_15240410_15240667_1 . . . 1136
27.251 so_18187297_18187846_2 . . . 1137
27.252 so_17536570_17567454_1 . . . 1137
27.253 so_41214250_41214250_2 . . . 1137
27.254 so_34246523_34247109_0 . . . 1137
27.255 so_15240410_15240667_2 . . . 1137
27.256 so_4171652_4171828_1 . . . 1137
27.257 so_19958679_19958840_0 . . . 1137
27.258 so_14329261_14329360_0 . . . 1137
27.259 so_14776416_14777308_0 . . . 1137
27.260 so_8049403_8049579_1 . . . 1137
27.261 so_8049403_8049579_3 . . . 1137
27.262 so_8049403_8049579_2 . . . 1137
27.263 so_15240410_15240667_3 . . . 1137

27.264 so_23751099_23751698_0 . . . 1138
27.265 so_23751099_23751631_1 . . . 1138
27.266 so_16540725_16540757_4 . . . 1138
27.267 so_3313527_3313543_0 . . . 1138
27.268 so_20700324_20701187_0 . . . 1138
27.269 so_748014_748059_1 . . . 1138
27.270 so_12774459_12774552_4 . . . 1138
27.271 so_40445685_40445875_2 . . . 1138
27.272 so_26756892_26757232_5 . . . 1138
27.273 so_26756892_26757232_2 . . . 1138
27.274 so_3535824_3535917_0 . . . 1138
27.275 so_40445685_40445875_3 . . . 1138
27.276 so_26756892_26757232_3 . . . 1138
27.277 so_27267546_27267758_1 . . . 1138
27.278 so_39256916_39257706_3 . . . 1138
27.279 so_39256916_39257706_2 . . . 1138
27.280 so_3535824_3535917_2 . . . 1138
27.281 so_15084007_15084813_0 . . . 1138
27.282 so_33042973_33043078_0 . . . 1138
27.283 so_39256916_39257706_4 . . . 1138
27.284 so_12774459_12774552_5 . . . 1138
27.285 so_19513203_19514276_0 . . . 1139
27.286 so_5036878_5036970_1 . . . 1139
27.287 so_7522479_7522537_0 . . . 1139
27.288 so_7522479_7522527_2 . . . 1139
27.289 so_4349778_4350012_2 . . . 1139
27.290 so_29720288_29720461_6 . . . 1140
27.291 so_4349778_4349778_3 . . . 1141
27.292 so_34266112_34266569_0 . . . 1141
27.293 so_12774459_12774552_1 . . . 1143
27.294 so_12774459_12774552_2 . . . 1143
27.295 so_30540287_30540674_7 . . . 1143
27.296 so_30540287_30540674_6 . . . 1143
27.297 so_12574594_12574606_0 . . . 1143
27.298 so_40445685_40445875_0 . . . 1144
27.299 so_40445685_40445875_1 . . . 1144
27.300 so_27267546_27267758_0 . . . 1144
27.301 so_12626414_12626698_0 . . . 1144
27.302 so_26887330_26887976_1 . . . 1144
27.303 so_32833632_32833712_1 . . . 1144
27.304 so_30540287_30540674_5 . . . 1145
27.305 so_24501731_24501759_0 . . . 1145
27.306 so_39256916_39257706_1 . . . 1145
27.307 so_26756892_26757232_0 . . . 1146
27.308 so_26756892_26757232_1 . . . 1146
27.309 so_32833632_32833712_2 . . . 1148
27.310 so_32833632_32833712_3 . . . 1148
27.311 so_27708629_29457433_1 . . . 1149
27.312 so_27708629_29457433_6 . . . 1149
27.313 so_27708629_29457433_3 . . . 1149
27.314 so_27708629_29457433_2 . . . 1149

28 Regular expressions library **1151**
28.1 General . . . 1151
28.2 Definitions . . . 1151
28.3 Requirements . . . 1152
28.4 so_33708892_33709665_1 . . . 1152
28.5 so_24018164_24018660_1 . . . 1152
28.6 so_4276207_4276232_1 . . . 1152
28.7 Header `<regex>` synopsis . . . 1154
28.8 Namespace `std::regex_constants` . . . 1161
28.9 so_47320905_47320905_0 . . . 1161
28.10 so_45909196_45910354_2 . . . 1161
28.11 so_29285322_29285323_7 . . . 1162
28.12 so_29787673_29788014_11 . . . 1162
28.13 so_29787673_29788014_12 . . . 1162
28.14 so_34017257_34017446_2 . . . 1162
28.15 so_37829849_37831395_3 . . . 1162
28.16 so_37829849_37831395_0 . . . 1162
28.17 so_27783017_27783506_0 . . . 1162
28.18 so_11870426_11870715_0 . . . 1162
28.19 so_45909196_45910354_5 . . . 1162
28.20 so_22840771_22840771_2 . . . 1164
28.21 so_36106154_36166656_1 . . . 1164
28.22 so_39712992_39714004_0 . . . 1164
28.23 so_23484952_23484952_0 . . . 1164
28.24 Class `regex_error` . . . 1164
28.25 so_36106154_36258868_0 . . . 1164
28.26 Class template `regex_traits` . . . 1164
28.27 so_11471932_11471932_0 . . . 1166
28.28 Class template `basic_regex` . . . 1167
28.29 so_30990841_30991076_0 . . . 1167
28.30 Class template `sub_match` . . . 1173
28.31 Class template `match_results` . . . 1179
28.32 so_36106154_36258868_2 . . . 1180
28.33 Regular expression algorithms . . . 1184
28.34 so_36106154_36258868_1 . . . 1184
28.35 so_36106154_36166656_0 . . . 1184
28.36 so_26696250_26696318_0 . . . 1184
28.37 so_25085615_25085615_2 . . . 1184
28.38 so_34017257_34017446_0 . . . 1186
28.39 so_25085615_25085615_1 . . . 1186
28.40 so_26696250_26696318_1 . . . 1186
28.41 so_12125057_12125696_1 . . . 1187
28.42 Regular expression iterators . . . 1189
28.43 so_21996955_21997128_1 . . . 1189
28.44 so_9797294_9797294_1 . . . 1192
28.45 Modified ECMAScript regular expression grammar . . . 1195
28.46 so_13230826_13230879_0 . . . 1195
28.47 so_23484952_23484952_1 . . . 1195
28.48 so_33708892_33709665_4 . . . 1197
28.49 so_33708892_33709665_3 . . . 1197
28.50 so_33708892_33709665_2 . . . 1197

28.51 so_33708892_33709665_5 . . . 1197
28.52 so_31994460_31995269_1 . . . 1197
28.53 so_31994460_31995269_0 . . . 1197

29 Atomic operations library 1199
29.1 General . . . 1199
29.2 Header `<atomic>` synopsis . . . 1199
29.3 Order and consistency . . . 1202
29.4 so_45972392_45973336_0 . . . 1202
29.5 so_47520748_47522708_2 . . . 1202
29.6 so_30691135_30692887_8 . . . 1202
29.7 so_39053600_39053600_2 . . . 1202
29.8 so_4213523_4213523_1 . . . 1202
29.9 so_15270432_15271217_2 . . . 1202
29.10 so_38677061_38680364_0 . . . 1203
29.11 so_47520748_47522708_1 . . . 1203
29.12 so_46065770_46069339_1 . . . 1203
29.13 so_39053600_42857017_2 . . . 1203
29.14 so_28783456_28845770_1 . . . 1203
29.15 so_21670459_21671069_0 . . . 1203
29.16 so_25478029_39056569_3 . . . 1203
29.17 so_39053600_42857017_4 . . . 1203
29.18 so_25290359_43506604_2 . . . 1204
29.19 so_30691135_30691135_0 . . . 1204
29.20 so_30691135_30692887_4 . . . 1204
29.21 so_16924525_16924525_1 . . . 1204
29.22 so_14791495_14792770_0 . . . 1204
29.23 so_43749985_43751510_0 . . . 1204
29.24 so_40272730_40273121_0 . . . 1204
29.25 so_25290359_43506604_1 . . . 1204
29.26 so_42003798_42011343_0 . . . 1204
29.27 so_32496236_32504997_0 . . . 1204
29.28 so_33956205_33957647_3 . . . 1204
29.29 so_20130571_20131290_0 . . . 1204
29.30 so_16924525_16924525_0 . . . 1204
29.31 Lock-free property . . . 1204
29.32 Atomic types . . . 1205
29.33 so_16355425_16355863_0 . . . 1207
29.34 so_20805336_20806026_0 . . . 1207
29.35 so_32694114_32694707_0 . . . 1207
29.36 so_38193204_38193204_0 . . . 1207
29.37 so_16876410_21311124_1 . . . 1207
29.38 so_16876410_16877779_0 . . . 1207
29.39 so_30254582_30869374_0 . . . 1207
29.40 so_35226128_35226186_2 . . . 1207
29.41 so_22493879_22494119_6 . . . 1207
29.42 so_23716391_23716762_7 . . . 1207
29.43 so_36320008_36320050_0 . . . 1207
29.44 so_35226128_35226186_3 . . . 1208
29.45 so_29761513_29761513_1 . . . 1208
29.46 so_29761513_29761513_0 . . . 1208

29.47 so_16423702_16423702_0 . . . 1208
29.48 so_29761513_29761584_0 . . . 1208
29.49 so_26164094_26164236_0 . . . 1208
29.50 Operations on atomic types . . . 1208
29.51 so_30254582_30869374_1 . . . 1209
29.52 so_11593927_11605751_0 . . . 1209
29.53 so_25332245_30877160_1 . . . 1209
29.54 so_19200859_19202791_0 . . . 1210
29.55 so_40367079_40884748_0 . . . 1210
29.56 so_36320008_36320050_2 . . . 1210
29.57 so_36320008_36320067_1 . . . 1210
29.58 so_13693871_13693871_9 . . . 1210
29.59 so_9552133_12919036_11 . . . 1210
29.60 so_19900524_19900680_0 . . . 1210
29.61 so_9552133_12919036_4 . . . 1211
29.62 so_20398094_20399497_1 . . . 1211
29.63 so_3732395_3732395_0 . . . 1211
29.64 so_9552133_12919036_5 . . . 1211
29.65 so_20398094_20399497_0 . . . 1211
29.66 so_21879331_21946549_4 . . . 1212
29.67 so_21879331_21881095_0 . . . 1212
29.68 so_25199838_25217283_0 . . . 1212
29.69 so_41130990_41131085_1 . . . 1213
29.70 so_25199838_25199838_1 . . . 1213
29.71 so_25199838_25199838_0 . . . 1213
29.72 so_25199838_25217283_1 . . . 1213
29.73 so_25199838_25217283_2 . . . 1213
29.74 so_25199838_25200031_0 . . . 1213
29.75 so_17914630_17914630_0 . . . 1213
29.76 so_26724418_26724770_1 . . . 1213
29.77 Flag type and operations . . . 1214
29.78 so_35226128_35226186_4 . . . 1214
29.79 so_20209021_20209226_0 . . . 1214
29.80 so_18304120_18304993_1 . . . 1214
29.81 so_24437396_24437665_1 . . . 1214
29.82 so_24437396_24438336_0 . . . 1214
29.83 so_24437396_24438336_1 . . . 1214
29.84 so_30691135_30692887_1 . . . 1215
29.85 Fences . . . 1215
29.86 so_4213523_4213523_5 . . . 1215
29.87 so_18223161_18430772_0 . . . 1215
29.88 so_38952761_38963036_0 . . . 1215
29.89 so_14167767_14183048_0 . . . 1215
29.90 so_17345680_17843102_0 . . . 1215

30 Thread support library **1217**
30.1 General . . . 1217
30.2 so_19759983_19760125_0 . . . 1217
30.3 Requirements . . . 1217
30.4 so_32659072_32659233_1 . . . 1217
30.5 so_14261013_14261013_0 . . . 1217

30.6 so_39355788_39355788_7 . . . 1217
30.7 so_29063585_29065296_2 . . . 1217
30.8 so_29063585_29065296_1 . . . 1217
30.9 so_42638847_42639024_1 . . . 1218
30.10 so_13142255_13142514_0 . . . 1218
30.11 so_42638847_42639024_2 . . . 1218
30.12 so_13142255_13142514_1 . . . 1218
30.13 so_29927695_29927766_0 . . . 1218
30.14 so_13142255_13142514_2 . . . 1218
30.15 so_22676859_22676925_0 . . . 1218
30.16 so_38252022_38253266_0 . . . 1218
30.17 so_20445051_20448763_1 . . . 1219
30.18 so_29437739_29438098_1 . . . 1219
30.19 so_29437739_29438098_2 . . . 1219
30.20 so_32770395_32770395_0 . . . 1219
30.21 so_25085615_25085615_3 . . . 1219
30.22 so_21059115_21059616_3 . . . 1220
30.23 so_28330471_28330636_2 . . . 1220
30.24 so_29693630_29694017_1 . . . 1220
30.25 Threads . . . 1220
30.26 so_40357447_40360142_1 . . . 1221
30.27 so_15445222_15445222_0 . . . 1221
30.28 so_40418833_40418962_0 . . . 1222
30.29 so_16064785_16065129_0 . . . 1222
30.30 so_24142110_24142110_0 . . . 1222
30.31 so_40418833_40418962_1 . . . 1222
30.32 so_40811399_40811706_1 . . . 1223
30.33 so_10001751_10001898_0 . . . 1223
30.34 so_16596950_16596950_1 . . . 1223
30.35 so_15080015_15080378_0 . . . 1223
30.36 so_19995534_19995691_1 . . . 1223
30.37 so_20313322_20313322_0 . . . 1223
30.38 so_10001751_10001898_1 . . . 1223
30.39 so_13813838_13814380_1 . . . 1223
30.40 so_24896622_24896622_0 . . . 1223
30.41 so_10001751_10001898_2 . . . 1223
30.42 so_21059115_21059616_2 . . . 1223
30.43 so_21059115_21059616_1 . . . 1223
30.44 so_15080015_15080378_1 . . . 1223
30.45 so_17655880_17655982_0 . . . 1223
30.46 so_28330471_28330636_0 . . . 1223
30.47 so_28330471_28330636_1 . . . 1223
30.48 so_19995534_19995691_3 . . . 1223
30.49 so_29693630_29694017_0 . . . 1223
30.50 so_18466924_18467058_0 . . . 1223
30.51 so_40811399_40811706_3 . . . 1223
30.52 so_18223161_18430772_2 . . . 1223
30.53 so_45203920_45227606_0 . . . 1223
30.54 so_17192991_17193859_0 . . . 1223
30.55 so_41427305_41427432_1 . . . 1223
30.56 so_46032965_46033613_0 . . . 1223

30.57 so_19995534_19995691_4 . . . 1223
30.58 so_32252143_32272900_0 . . . 1223
30.59 so_27388068_27388981_9 . . . 1223
30.60 so_14408796_14409088_0 . . . 1223
30.61 so_25559918_25559940_0 . . . 1223
30.62 so_34330576_34331455_1 . . . 1223
30.63 so_13983984_13984169_2 . . . 1223
30.64 so_13983984_13984169_1 . . . 1223
30.65 so_27392743_27392989_0 . . . 1223
30.66 so_26041109_26041203_0 . . . 1223
30.67 so_26826806_26826859_0 . . . 1223
30.68 so_30862138_30862784_0 . . . 1223
30.69 so_22905207_22905277_0 . . . 1223
30.70 so_21197912_21197942_2 . . . 1223
30.71 so_37678354_37679754_1 . . . 1223
30.72 so_24080167_24083086_0 . . . 1223
30.73 so_13999432_14000704_2 . . . 1223
30.74 so_13999432_14000704_3 . . . 1223
30.75 so_30309174_30309285_2 . . . 1224
30.76 so_27431916_27433549_2 . . . 1224
30.77 so_39925092_39925402_0 . . . 1224
30.78 so_27431916_27433549_3 . . . 1224
30.79 so_46032965_46033613_1 . . . 1224
30.80 so_27109314_27109410_0 . . . 1224
30.81 so_39353607_39353607_3 . . . 1224
30.82 so_38107707_38109735_4 . . . 1224
30.83 so_39355788_39355788_2 . . . 1224
30.84 so_38107707_38109735_2 . . . 1224
30.85 so_38919108_38919187_0 . . . 1224
30.86 so_14402328_14404498_0 . . . 1225
30.87 so_42638847_42639024_0 . . . 1226
30.88 so_46258641_46259362_1 . . . 1226
30.89 so_30538508_30539043_0 . . . 1226
30.90 Mutual exclusion . . . 1226
30.91 so_20445051_20448763_0 . . . 1227
30.92 so_45475241_45477313_2 . . . 1227
30.93 so_22703439_22703633_0 . . . 1227
30.94 so_22703439_22703633_1 . . . 1227
30.95 so_39353607_39353607_5 . . . 1227
30.96 so_39355788_39355788_4 . . . 1227
30.97 so_39353607_39353674_0 . . . 1227
30.98 so_39353607_39353607_1 . . . 1227
30.99 so_22203168_22203335_0 . . . 1227
30.100 so_17551256_17551739_0 . . . 1227
30.101 so_39355788_39355788_0 . . . 1227
30.102 so_10949863_10950183_0 . . . 1227
30.103 so_31205040_31205382_1 . . . 1227
30.104 so_43487357_43487579_1 . . . 1227
30.105 so_39353607_39353674_1 . . . 1227
30.106 so_6804044_6804113_2 . . . 1227
30.107 so_44652353_44652576_1 . . . 1227

30.108 so_16664375_16664558_1 . . . 1227
30.109 so_22203168_22203335_1 . . . 1227
30.110 so_27257440_27257523_1 . . . 1227
30.111 so_39353607_39353607_7 . . . 1228
30.112 so_39353607_39353607_6 . . . 1228
30.113 so_39355788_39357753_2 . . . 1228
30.114 so_39355788_39355788_5 . . . 1228
30.115 so_39355788_39355788_6 . . . 1228
30.116 so_28005135_28021381_0 . . . 1228
30.117 so_33907999_33915524_0 . . . 1228
30.118 so_33907999_33915524_1 . . . 1228
30.119 so_43487357_43487579_0 . . . 1228
30.120 so_44652353_44652576_0 . . . 1228
30.121 so_10949863_10950183_2 . . . 1228
30.122 so_27257440_27257523_0 . . . 1228
30.123 so_44652353_44652576_2 . . . 1228
30.124 so_44652353_44652576_4 . . . 1228
30.125 so_10949863_10950183_3 . . . 1228
30.126 so_31750036_31753894_0 . . . 1229
30.127 so_31205040_31205382_0 . . . 1229
30.128 so_31205040_31205418_0 . . . 1229
30.129 so_40919654_40920010_2 . . . 1229
30.130 so_16664375_16664420_1 . . . 1229
30.131 so_17551256_17551739_1 . . . 1229
30.132 so_39355788_39357753_0 . . . 1229
30.133 so_43487357_43487468_0 . . . 1229
30.134 so_25495413_25496464_0 . . . 1229
30.135 so_37203741_37203741_0 . . . 1230
30.136 so_16664375_16664420_0 . . . 1232
30.137 so_13359582_13359813_0 . . . 1233
30.138 so_41847511_41850904_0 . . . 1233
30.139 so_10949863_10950183_1 . . . 1233
30.140 so_18937564_18937820_0 . . . 1233
30.141 so_15313296_15313944_9 . . . 1233
30.142 so_15313296_15313296_2 . . . 1233
30.143 so_13901136_13901160_0 . . . 1235
30.144 so_40029838_40030219_0 . . . 1237
30.145 so_17806697_17807273_0 . . . 1237
30.146 so_14920997_14921057_0 . . . 1237
30.147 so_18520983_18520983_1 . . . 1244
30.148 so_18520983_18521108_0 . . . 1244
30.149 so_18520983_18520983_2 . . . 1244
30.150 so_18520983_18521272_0 . . . 1244
30.151 so_15311469_15311469_0 . . . 1245
30.152 so_43054677_43055483_0 . . . 1245
30.153 so_17407553_17408879_1 . . . 1245
30.154 so_9964786_9964844_1 . . . 1245
30.155 so_43054677_43055483_1 . . . 1245
30.156 so_9964786_9964844_2 . . . 1245
30.157 Condition variables . . . 1246
30.158 so_31871267_31874630_1 . . . 1246

30.159 so_8758353_8758379_0 . . . 1246
30.160 so_17984552_17988574_0 . . . 1246
30.161 so_31871267_31874630_3 . . . 1247
30.162 so_25796287_25796860_5 . . . 1248
30.163 so_25796287_25796860_2 . . . 1248
30.164 so_14582505_14582643_1 . . . 1248
30.165 so_14582505_14582643_0 . . . 1248
30.166 so_21734849_21735176_0 . . . 1248
30.167 so_19829754_19832374_0 . . . 1248
30.168 so_15084149_15084225_0 . . . 1248
30.169 so_21734849_21735176_1 . . . 1248
30.170 so_13556763_13556919_0 . . . 1248
30.171 so_24846995_24847058_0 . . . 1248
30.172 so_24846995_24847058_2 . . . 1248
30.173 so_24846995_24847058_1 . . . 1248
30.174 so_34120422_34121629_1 . . . 1248
30.175 so_46088363_46142373_0 . . . 1248
30.176 so_15278343_15278455_0 . . . 1249
30.177 so_15278343_15278455_2 . . . 1249
30.178 so_29437739_29438098_0 . . . 1251
30.179 Futures . . . 1253
30.180 so_47202990_47219655_6 . . . 1255
30.181 so_47202990_47219655_7 . . . 1255
30.182 so_26278797_26278926_0 . . . 1256
30.183 so_38141229_38141497_4 . . . 1256
30.184 so_38141229_38141560_6 . . . 1256
30.185 so_38141229_38141560_7 . . . 1256
30.186 so_38141229_38141560_8 . . . 1256
30.187 so_38141229_38141560_5 . . . 1256
30.188 so_38141229_38141497_7 . . . 1256
30.189 so_36816928_36817064_4 . . . 1256
30.190 so_36816928_36817064_6 . . . 1256
30.191 so_29026607_29027437_0 . . . 1256
30.192 so_25803403_25803403_9 . . . 1257
30.193 so_19111028_19111069_2 . . . 1257
30.194 so_28069443_38524181_1 . . . 1257
30.195 so_27423497_27423497_2 . . . 1259
30.196 so_27423497_27423497_1 . . . 1259
30.197 so_22664296_22665367_4 . . . 1259
30.198 so_38141229_38141560_2 . . . 1260
30.199 so_32333219_32333219_2 . . . 1261
30.200 so_33899615_33900775_0 . . . 1261
30.201 so_22461249_22461668_0 . . . 1261
30.202 so_12811426_12814921_0 . . . 1261
30.203 so_12811426_12814921_1 . . . 1261
30.204 so_41371714_41416686_1 . . . 1262
30.205 so_32333219_32333444_0 . . . 1262
30.206 so_32333219_32333219_0 . . . 1263
30.207 so_32333219_32333444_2 . . . 1264
30.208 so_42004877_42004877_7 . . . 1265
30.209 so_41955493_41955881_0 . . . 1265

30.210 so_47436919_47451933_0 . . . 1265
30.211 so_9566187_36798462_0 . . . 1265
30.212 so_13813838_13814380_2 . . . 1265
30.213 so_47573564_47574759_0 . . . 1265
30.214 so_3672563_3672617_2 . . . 1265
30.215 so_42004877_42004877_10 . . . 1265
30.216 so_42004877_42007403_1 . . . 1265
30.217 so_42004877_42007403_0 . . . 1265
30.218 so_22461249_22461668_1 . . . 1265
30.219 so_9355110_9355110_0 . . . 1265
30.220 so_36766590_36766590_0 . . . 1265
30.221 so_11004273_11004455_1 . . . 1265
30.222 so_12834281_12834871_0 . . . 1265
30.223 so_12834281_12834871_1 . . . 1265
30.224 so_42754244_42754244_0 . . . 1265
30.225 so_34840754_34841184_0 . . . 1266
30.226 so_25673475_25673947_2 . . . 1266
30.227 so_34500067_34500099_0 . . . 1266
30.228 so_25698634_25698913_0 . . . 1266
30.229 so_9733432_9733751_0 . . . 1266
30.230 so_26278797_26279344_0 . . . 1266
30.231 so_26278797_26279344_1 . . . 1266
30.232 so_38141229_38141497_2 . . . 1266
30.233 so_38141229_38141497_0 . . . 1266
30.234 so_28148460_28148540_0 . . . 1266
30.235 so_28148460_28148540_1 . . . 1266
30.236 so_22664296_22665367_2 . . . 1266
30.237 so_42004877_42004877_2 . . . 1266
30.238 so_42004877_42004877_4 . . . 1266
30.239 so_47202990_47219655_1 . . . 1266
30.240 so_17493928_17494800_0 . . . 1267
30.241 so_28069443_28069443_0 . . . 1268
30.242 so_12834281_12834899_0 . . . 1268
30.243 so_24547203_24549996_1 . . . 1268
30.244 so_47202990_47219655_2 . . . 1268
30.245 so_17493928_17494800_1 . . . 1268
30.246 so_24547203_24549996_2 . . . 1268
30.247 so_47202990_47219655_4 . . . 1269

A Grammar summary **1270**
A.1 Keywords . . . 1270
A.2 Lexical conventions . . . 1270
A.3 Basic concepts . . . 1275
A.4 Expressions . . . 1275
A.5 Statements . . . 1278
A.6 Declarations . . . 1279
A.7 Declarators . . . 1283
A.8 Classes . . . 1285
A.9 Derived classes . . . 1285
A.10 Special member functions . . . 1286
A.11 Overloading . . . 1286

A.12 Templates . . . 1286
A.13 Exception handling . . . 1287
A.14 Preprocessing directives . . . 1288

B Implementation quantities **1290**
B.1 so_6589946_6590027_0 . . . 1290
B.2 so_4582012_4582026_0 . . . 1290
B.3 so_14603752_14604018_0 . . . 1290
B.4 so_45930605_45930788_5 . . . 1290
B.5 so_20355948_20356120_0 . . . 1290
B.6 so_20355948_20356073_0 . . . 1290
B.7 so_12701170_12704054_1 . . . 1290
B.8 so_20405321_20405380_0 . . . 1290
B.9 so_29218872_29218954_0 . . . 1290
B.10 so_9258525_9259119_1 . . . 1290
B.11 so_10087113_10087264_1 . . . 1290
B.12 so_10087113_10087135_0 . . . 1290
B.13 so_18676854_18676986_0 . . . 1290
B.14 so_757270_757385_1 . . . 1290
B.15 so_757270_757385_2 . . . 1290
B.16 so_4582012_4582026_1 . . . 1290
B.17 so_14603752_14604018_1 . . . 1290
B.18 so_14603752_14604018_4 . . . 1290
B.19 so_14603752_14604018_3 . . . 1290
B.20 so_14603752_14604018_2 . . . 1290
B.21 so_24646221_24646440_1 . . . 1290
B.22 so_4696553_4696585_2 . . . 1290
B.23 so_45930605_45930788_2 . . . 1290
B.24 so_45930605_45930788_8 . . . 1290
B.25 so_45930605_45930788_6 . . . 1290
B.26 so_37536573_37536718_1 . . . 1290
B.27 so_35971705_35971705_0 . . . 1290
B.28 so_46961876_46961990_0 . . . 1290
B.29 so_20355948_20356054_2 . . . 1290
B.30 so_20355948_20356073_2 . . . 1290
B.31 so_20355948_20356054_3 . . . 1290
B.32 so_20355948_20356054_1 . . . 1290
B.33 so_20355948_20356120_1 . . . 1290
B.34 so_20355948_20356073_1 . . . 1290
B.35 so_11394924_11396061_2 . . . 1290
B.36 so_10930091_10930245_0 . . . 1290

C Compatibility **1292**
C.1 C++ and ISO C . . . 1292
C.2 C++ and ISO C++ 2003 . . . 1301
C.3 C++ and ISO C++ 2011 . . . 1308
C.4 C standard library . . . 1309
C.5 so_17504565_17504565_7 . . . 1309
C.6 so_17504565_17504565_8 . . . 1310
C.7 so_4404725_4404753_1 . . . 1310
C.8 so_9149368_9149374_1 . . . 1310

C.9 so_7131858_7132065_4 . . . 1310
C.10 so_43011737_43011792_1 . . . 1310
C.11 so_13889467_43016708_1 . . . 1310
C.12 so_15616177_15616226_1 . . . 1311
C.13 so_32136092_32136460_4 . . . 1311
C.14 so_651361_651380_1 . . . 1311
C.15 so_651361_651378_2 . . . 1311
C.16 so_34725939_34726797_2 . . . 1311
C.17 so_34725939_34726289_4 . . . 1311
C.18 so_34725939_34726289_2 . . . 1311
C.19 so_34725939_34726797_0 . . . 1311
C.20 so_15836499_15836511_6 . . . 1311
C.21 so_19083709_19083981_2 . . . 1311
C.22 so_9003425_9003576_0 . . . 1311
C.23 so_21909837_21910159_2 . . . 1311
C.24 so_7200674_7200762_1 . . . 1311
C.25 so_25650592_25656153_0 . . . 1311
C.26 so_25650592_25656153_1 . . . 1311
C.27 so_32302615_32469629_0 . . . 1311
C.28 so_40395066_40396015_0 . . . 1311
C.29 so_18775044_18775126_3 . . . 1311
C.30 so_33020532_33020942_0 . . . 1311
C.31 so_29422970_29423564_0 . . . 1311
C.32 so_27951344_27951801_0 . . . 1311
C.33 so_9169453_9169978_1 . . . 1312

D Compatibility features **1313**
D.1 so_34139882_34140653_1 . . . 1313
D.2 so_34139882_34140653_3 . . . 1313
D.3 Increment operator with `bool` operand . . . 1313
D.4 `register` keyword . . . 1313
D.5 so_15908835_15908876_2 . . . 1313
D.6 Implicit declaration of copy functions . . . 1313
D.7 so_4782757_27891048_0 . . . 1313
D.8 Dynamic exception specifications . . . 1313
D.9 C standard library headers . . . 1313
D.10 so_43011737_43011752_0 . . . 1313
D.11 so_13642827_13642921_0 . . . 1313
D.12 so_30875069_30875579_2 . . . 1313
D.13 so_10460250_10460277_0 . . . 1313
D.14 so_8380805_8380835_3 . . . 1313
D.15 so_31800065_31800404_1 . . . 1313
D.16 so_29897661_29900474_1 . . . 1313
D.17 so_2649207_2649315_3 . . . 1313
D.18 so_28696522_28696670_3 . . . 1313
D.19 so_10466096_10469611_0 . . . 1313
D.20 so_10466096_10466167_0 . . . 1313
D.21 so_5539249_37438120_1 . . . 1313
D.22 so_16518210_16518251_0 . . . 1313
D.23 so_4404725_4404753_0 . . . 1313
D.24 so_9149368_9149374_0 . . . 1313

D.25 so_19587434_19588293_0 . . . 1313
D.26 so_10625716_10626042_0 . . . 1313
D.27 so_4405887_4405887_0 . . . 1313
D.28 so_33101264_33101406_1 . . . 1313
D.29 so_32606023_32606271_0 . . . 1313
D.30 so_13642827_13642921_2 . . . 1313
D.31 so_13040139_13040283_1 . . . 1313
D.32 so_26791436_26791436_0 . . . 1313
D.33 so_5135734_5136535_0 . . . 1313
D.34 so_1524139_1524387_1 . . . 1313
D.35 so_6215467_6215534_0 . . . 1313
D.36 so_6215467_6215534_2 . . . 1313
D.37 so_39130040_39132787_0 . . . 1313
D.38 so_10460250_10460277_2 . . . 1313
D.39 so_8380805_8380835_5 . . . 1313
D.40 so_18128899_18128988_0 . . . 1313
D.41 so_15604129_15604270_1 . . . 1313
D.42 so_29551755_29551827_0 . . . 1314
D.43 so_10625716_10626042_1 . . . 1314
D.44 so_13642827_13642921_3 . . . 1314
D.45 so_13040139_13040283_2 . . . 1314
D.46 so_26791436_26791436_1 . . . 1314
D.47 so_6215467_6215534_3 . . . 1314
D.48 so_6215467_6215534_1 . . . 1314
D.49 so_39130040_39132787_1 . . . 1314
D.50 so_10460250_10460277_3 . . . 1314
D.51 so_8380805_8380835_6 . . . 1314
D.52 so_18128899_18128988_1 . . . 1314
D.53 Old iostreams members . . . 1314
D.54 `char*` streams . . . 1315
D.55 so_24482028_24482042_1 . . . 1317
D.56 so_28950575_28950796_0 . . . 1318
D.57 Function objects . . . 1324
D.58 Binders . . . 1327
D.59 `auto_ptr` . . . 1328
D.60 so_111478_13141169_1 . . . 1329
D.61 so_2576766_2576775_0 . . . 1330
D.62 Violating *exception-specifications* . . . 1331
D.63 Random shuffle . . . 1332
D.64 so_16967828_16967892_0 . . . 1332
D.65 so_16967828_16967892_1 . . . 1332

E Universal character names for identifier characters 1333
E.1 Ranges of characters allowed . . . 1333
E.2 Ranges of characters disallowed initially . . . 1333

F Cross references 1334

List of Tables

1 Trigraph sequences . . . 20
2 Alternative tokens . . . 22
3 Identifiers with special meaning . . . 24
4 Keywords . . . 25
5 Alternative representations . . . 25
6 Types of integer literals . . . 27
7 Escape sequences . . . 29
8 String literal concatenations . . . 32

9 Relations on `const` and `volatile` . . . 84

10 *simple-type-specifier*s and the types they specify . . . 177

11 Relationship between operator and function call notation . . . 335
12 Conversions . . . 344

13 Library categories . . . 474
14 C++ library headers . . . 485
15 C++ headers for C library facilities . . . 485
16 C++ headers for freestanding implementations . . . 486
17 `EqualityComparable` requirements . . . 488
18 `LessThanComparable` requirements . . . 488
19 `DefaultConstructible` requirements . . . 488
20 `MoveConstructible` requirements . . . 488
21 `CopyConstructible` requirements (in addition to `MoveConstructible`) . . . 489
22 `MoveAssignable` requirements . . . 489
23 `CopyAssignable` requirements (in addition to `MoveAssignable`) . . . 489
24 `Destructible` requirements . . . 489
25 `NullablePointer` requirements . . . 491
26 `Hash` requirements . . . 492
27 Descriptive variable definitions . . . 492
28 Allocator requirements . . . 493

29 Language support library summary . . . 507
30 Header `<cstddef>` synopsis . . . 507
31 Header `<climits>` synopsis . . . 517
32 Header `<cfloat>` synopsis . . . 518
33 Header `<cstdlib>` synopsis . . . 519
34 Header `<csetjmp>` synopsis . . . 538
35 Header `<csignal>` synopsis . . . 538
36 Header `<cstdalign>` synopsis . . . 538
37 Header `<cstdarg>` synopsis . . . 538
38 Header `<cstdbool>` synopsis . . . 538
39 Header `<cstdlib>` synopsis . . . 538
40 Header `<ctime>` synopsis . . . 538

41 Diagnostics library summary . . . 539
42 Header `<cassert>` synopsis . . . 543
43 Header `<cerrno>` synopsis . . . 544

44 General utilities library summary . . . 555
45 Header `<cstdlib>` synopsis . . . 597
46 Header `<cstring>` synopsis . . . 598
47 Primary type category predicates . . . 651
48 Composite type category predicates . . . 652
49 Type property predicates . . . 653
50 Type property queries . . . 659
51 Type relationship predicates . . . 660
52 Const-volatile modifications . . . 661
53 Reference modifications . . . 662
54 Sign modifications . . . 662
55 Array modifications . . . 663
56 Pointer modifications . . . 663
57 Other transformations . . . 664
58 Expressions used to perform ratio arithmetic . . . 668
59 Clock requirements . . . 672
60 Header `<ctime>` synopsis . . . 685

61 Strings library summary . . . 693
62 Character traits requirements . . . 694
63 `basic_string(const Allocator&)` effects . . . 708
64 `basic_string(const basic_string&)` effects . . . 708
65 `basic_string(const basic_string&, size_type, size_type, const Allocator&)` effects 709
66 `basic_string(const charT*, size_type, const Allocator&)` effects . . . 709
67 `basic_string(const charT*, const Allocator&)` effects . . . 709
68 `basic_string(size_t, charT, const Allocator&)` effects . . . 710
69 `basic_string(const basic_string&, const Allocator&)` and `basic_string(basic_string&&, const Allocator&)` effects . . . 710
70 `operator=(const basic_string&)` effects . . . 711
71 `operator=(basic_string&&)` effects . . . 711
72 `compare()` results . . . 725
73 Potential `mbstate_t` data races . . . 734
74 Header `<cctype>` synopsis . . . 734
75 Header `<cwctype>` synopsis . . . 735
76 Header `<cstring>` synopsis . . . 735
77 Header `<cwchar>` synopsis . . . 735
78 Header `<cstdlib>` synopsis . . . 735
79 Header `<cuchar>` synopsis . . . 736

80 Localization library summary . . . 737
81 Locale category facets . . . 741
82 Required specializations . . . 742
83 `do_in/do_out` result values . . . 760
84 `do_unshift` result values . . . 760
85 Integer conversions . . . 764
86 Length modifier . . . 765
87 Integer conversions . . . 768

88 Floating-point conversions . . . 769
89 Length modifier . . . 769
90 Numeric conversions . . . 769
91 Fill padding . . . 770
92 `do_get_date` effects . . . 777
93 Header `<clocale>` synopsis . . . 792
94 Potential `setlocale` data races . . . 792

95 Containers library summary . . . 793
96 Container requirements . . . 794
97 Reversible container requirements . . . 797
98 Optional container operations . . . 798
99 Allocator-aware container requirements . . . 799
100 Sequence container requirements (in addition to container) . . . 801
101 Optional sequence container operations . . . 804
102 Associative container requirements (in addition to container) . . . 806
103 Unordered associative container requirements (in addition to container) . . . 814

104 Iterators library summary . . . 900
105 Relations among iterator categories . . . 900
106 Iterator requirements . . . 902
107 Input iterator requirements (in addition to Iterator) . . . 902
108 Output iterator requirements (in addition to Iterator) . . . 903
109 Forward iterator requirements (in addition to input iterator) . . . 905
110 Bidirectional iterator requirements (in addition to forward iterator) . . . 905
111 Random access iterator requirements (in addition to bidirectional iterator) . . . 906

112 Algorithms library summary . . . 935
113 Header `<cstdlib>` synopsis . . . 973

114 Numerics library summary . . . 975
115 Seed sequence requirements . . . 989
116 Uniform random number generator requirements . . . 990
117 Random number engine requirements . . . 991
118 Random number distribution requirements . . . 995
119 Header `<cmath>` synopsis . . . 1057
120 Header `<cstdlib>` synopsis . . . 1057

121 Input/output library summary . . . 1062
122 `fmtflags` effects . . . 1072
123 `fmtflags` constants . . . 1072
124 `iostate` effects . . . 1072
125 `openmode` effects . . . 1073
126 `seekdir` effects . . . 1073
127 Position type requirements . . . 1077
128 `basic_ios::init()` effects . . . 1080
129 `basic_ios::copyfmt()` effects . . . 1081
130 `seekoff` positioning . . . 1129
131 `newoff` values . . . 1129
132 File open modes . . . 1139
133 `seekoff` effects . . . 1141

134 Header `<cstdio>` synopsis . . . 1149
135 Header `<cinttypes>` synopsis . . . 1149

136 Regular expressions library summary . . . 1151
137 Regular expression traits class requirements . . . 1152
138 `syntax_option_type` effects . . . 1162
139 `regex_constants::match_flag_type` effects when obtaining a match against a character container sequence `[first,last)`. . . . 1163
140 `error_type` values in the C locale . . . 1164
141 Character class names and corresponding `ctype` masks . . . 1168
142 `match_results` assignment operator effects . . . 1181
143 Effects of `regex_match` algorithm . . . 1185
144 Effects of `regex_search` algorithm . . . 1186

145 Atomics library summary . . . 1199
146 `atomic` integral typedefs . . . 1208
147 `atomic <inttypes.h>` typedefs . . . 1209
148 Atomic arithmetic computations . . . 1213

149 Thread support library summary . . . 1217

150 Standard macros . . . 1309
151 Standard values . . . 1309
152 Standard types . . . 1309
153 Standard structs . . . 1310
154 Standard functions . . . 1310

155 C headers . . . 1313
156 `strstreambuf(streamsize)` effects . . . 1317
157 `strstreambuf(void* (*)(size_t), void (*)(void*))` effects . . . 1317
158 `strstreambuf(charT*, streamsize, charT*)` effects . . . 1318
159 `seekoff` positioning . . . 1320
160 `newoff` values . . . 1320

List of Figures

1 Expression category taxonomy 85

2 Directed acyclic graph 266
3 Non-virtual base 267
4 Virtual base 268
5 Virtual and non-virtual base 268
6 Name lookup 271

7 Stream position, offset, and size types [non-normative] 1062

1 General [intro]

1.1 Scope [intro.scope]

1 This International Standard specifies requirements for implementations of the C++ programming language. The first such requirement is that they implement the language, and so this International Standard also defines C++. Other requirements and relaxations of the first requirement appear at various places within this International Standard.

2 C++ is a general purpose programming language based on the C programming language as described in ISO/IEC 9899:1999 *Programming languages — C* (hereinafter referred to as the *C standard*). In addition to the facilities provided by C, C++ provides additional data types, classes, templates, exceptions, namespaces, operator overloading, function name overloading, references, free store management operators, and additional library facilities.

1.4 Normative references [intro.refs]

1 The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

— Ecma International, *ECMAScript Language Specification*, Standard Ecma-262, third edition, 1999.

— ISO/IEC 2382 (all parts), *Information technology — Vocabulary*

— ISO/IEC 9899:1999, *Programming languages — C*

— ISO/IEC 9899:1999/Cor.1:2001(E), *Programming languages — C, Technical Corrigendum 1*

— ISO/IEC 9899:1999/Cor.2:2004(E), *Programming languages — C, Technical Corrigendum 2*

— ISO/IEC 9899:1999/Cor.3:2007(E), *Programming languages — C, Technical Corrigendum 3*

— ISO/IEC 9945:2003, *Information Technology — Portable Operating System Interface (POSIX)*

— ISO/IEC 10646-1:1993, *Information technology — Universal Multiple-Octet Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane*

— ISO/IEC TR 19769:2004, *Information technology — Programming languages, their environments and system software interfaces — Extensions for the programming language C to support new character data types*

1 The library described in Clause 7 of ISO/IEC 9899:1999 and Clause 7 of ISO/IEC 9899:1999/Cor.1:2001 and Clause 7 of ISO/IEC 9899:1999/Cor.2:2003 is hereinafter called the *C standard library*.[1]

1 The library described in ISO/IEC TR 19769:2004 is hereinafter called the *C Unicode TR*.

2 The operating system interface described in ISO/IEC 9945:2003 is hereinafter called *POSIX*.

1) With the qualifications noted in Clauses 18 through 30 and in C.4, the C standard library is a subset of the C++ standard library.

3 The ECMAScript Language Specification described in Standard Ecma-262 is hereinafter called *ECMA-262*.

1.27 Terms and definitions [intro.defs]

1 For the purposes of this document, the following definitions apply.

2 17.9 defines additional terms that are used only in Clauses 17 through 30 and Annex D.

3

Terms that are used only in a small portion of this International Standard are defined where they are used and italicized where they are defined.

1.195.1 **[defns.argument]**
argument
<function call expression> expression in the comma-separated list bounded by the parentheses

1.195.2 **[defns.argument.macro]**
argument
<function-like macro> sequence of preprocessing tokens in the comma-separated list bounded by the parentheses

1.195.3 **[defns.argument.throw]**
argument
<throw expression> the operand of `throw`

1.195.4 **[defns.argument.templ]**
argument
<template instantiation> expression, *type-id* or *template-name* in the comma-separated list bounded by the angle brackets

1.195.5 **[defns.cond.supp]**
conditionally-supported
program construct that an implementation is not required to support
[*Note:* Each implementation documents all conditionally-supported constructs that it does not support. — *end note*]

1.195.6 **[defns.diagnostic]**
diagnostic message
message belonging to an implementation-defined subset of the implementation's output messages

1.195.7 **[defns.dynamic.type]**
dynamic type
<glvalue> type of the most derived object (1.319) to which the glvalue denoted by a glvalue expression refers
[*Example:* if a pointer (8.94.1) `p` whose static type is "pointer to class `B`" is pointing to an object of class `D`, derived from `B` (Clause 10), the dynamic type of the expression `*p` is "`D`." References (8.97.1) are treated similarly. — *end example*]

1.195.8 **[defns.dynamic.type.prvalue]**
dynamic type
<prvalue> static type of the prvalue expression

1.195.9 **[defns.ill.formed]**
ill-formed program
program that is not well formed

1.195.10 **[defns.impl.defined]**
implementation-defined behavior
behavior, for a well-formed program construct and correct data, that depends on the implementation and that each implementation documents

1.195.11 **[defns.impl.limits]**
implementation limits
restrictions imposed upon programs by the implementation

1.195.12 **[defns.locale.specific]**
locale-specific behavior
behavior that depends on local conventions of nationality, culture, and language that each implementation documents

1.195.13 **[defns.multibyte]**
multibyte character
sequence of one or more bytes representing a member of the extended character set of either the source or the execution environment
[*Note:* The extended character set is a superset of the basic character set (2.154). — *end note*]

1.195.14 **[defns.parameter]**
parameter
<function or catch clause> object or reference declared as part of a function declaration or definition or in the catch clause of an exception handler that acquires a value on entry to the function or handler

1.195.15 **[defns.parameter.macro]**
parameter
<function-like macro> identifier from the comma-separated list bounded by the parentheses immediately following the macro name

1.195.16 **[defns.parameter.templ]**
parameter
<template> *template-parameter*

1.195.17 **[defns.signature]**
signature
<function> name, parameter type list (8.360.1), and enclosing namespace (if any)
[*Note:* Signatures are used as a basis for name mangling and linking. — *end note*]

1.195.18 **[defns.signature.templ]**

signature

<function template> name, parameter type list (8.360.1), enclosing namespace (if any), return type, and template parameter list

1.195.19 **[defns.signature.spec]**

signature

<function template specialization> signature of the template of which it is a specialization and its template arguments (whether explicitly specified or deduced)

1.195.20 **[defns.signature.member]**

signature

<class member function> name, parameter type list (8.360.1), class of which the function is a member, *cv*-qualifiers (if any), and *ref-qualifier* (if any)

1.195.21 **[defns.signature.member.templ]**

signature

<class member function template> name, parameter type list (8.360.1), class of which the function is a member, *cv*-qualifiers (if any), *ref-qualifier* (if any), return type, and template parameter list

1.195.22 **[defns.signature.member.spec]**

signature

<class member function template specialization> signature of the member function template of which it is a specialization and its template arguments (whether explicitly specified or deduced)

1.195.23 **[defns.static.type]**

static type

type of an expression (3.2955) resulting from analysis of the program without considering execution semantics [*Note:* The static type of an expression depends only on the form of the program in which the expression appears, and does not change while the program is executing. — *end note*]

1.195.24 **[defns.undefined]**

undefined behavior

behavior for which this International Standard imposes no requirements

[*Note:* Undefined behavior may be expected when this International Standard omits any explicit definition of behavior or when a program uses an erroneous construct or erroneous data. Permissible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message). Many erroneous program constructs do not engender undefined behavior; they are required to be diagnosed. — *end note*]

1.195.25 **[defns.unspecified]**

unspecified behavior

behavior, for a well-formed program construct and correct data, that depends on the implementation

[*Note:* The implementation is not required to document which behavior occurs. The range of possible behaviors is usually delineated by this International Standard. — *end note*]

1.195.26 **[defns.well.formed]**

well-formed program

C++ program constructed according to the syntax rules, diagnosable semantic rules, and the One Definition Rule (3.91).

1.196 Implementation compliance [intro.compliance]

1

The set of *diagnosable rules* consists of all syntactic and semantic rules in this International Standard except for those rules containing an explicit notation that "no diagnostic is required" or which are described as resulting in "undefined behavior."

1

Although this International Standard states only requirements on C++ implementations, those requirements are often easier to understand if they are phrased as requirements on programs, parts of programs, or execution of programs. Such requirements have the following meaning:

— If a program contains no violations of the rules in this International Standard, a conforming implementation shall, within its resource limits, accept and correctly execute[2] that program.

— If a program contains a violation of any diagnosable rule or an occurrence of a construct described in this Standard as "conditionally-supported" when the implementation does not support that construct, a conforming implementation shall issue at least one diagnostic message.

— If a program contains a violation of a rule for which no diagnostic is required, this International Standard places no requirement on implementations with respect to that program.

1

For classes and class templates, the library Clauses specify partial definitions. Private members (Clause 11) are not specified, but each implementation shall supply them to complete the definitions according to the description in the library Clauses.

1

For functions, function templates, objects, and values, the library Clauses specify declarations. Implementations shall supply definitions consistent with the descriptions in the library Clauses.

1 The names defined in the library have namespace scope (7.1142). A C++ translation unit (2.6) obtains access to these names by including the appropriate standard library header (16.48).

2

The templates, classes, functions, and objects in the library have external linkage (3.1459). The implementation provides definitions for standard library entities, as necessary, while combining translation units to form a complete C++ program (2.6).

1

Two kinds of implementations are defined: a *hosted implementation* and a *freestanding implementation*. For a hosted implementation, this International Standard defines the set of available libraries. A freestanding implementation is one in which execution may take place without the benefit of an operating system, and has an implementation-defined set of libraries that includes certain language-support libraries (17.97.0.1).

1

2) "Correct execution" can include undefined behavior, depending on the data being processed; see 1.27 and 1.430.

A conforming implementation may have extensions (including additional library functions), provided they do not alter the behavior of any well-formed program. Implementations are required to diagnose programs that use such extensions that are ill-formed according to this International Standard. Having done so, however, they can compile and execute such programs.

1

Each implementation shall include documentation that identifies all conditionally-supported constructs that it does not support and defines all locale-specific characteristics.[3]

1.273 Structure of this International Standard [intro.structure]

1 Clauses 2 through 16 describe the C++ programming language. That description includes detailed syntactic specifications in a form described in 1.274. For convenience, Annex A repeats all such syntactic specifications.

2 Clauses 18 through 30 and Annex D (the *library clauses*) describe the Standard C++ library. That description includes detailed descriptions of the templates, classes, functions, constants, and macros that constitute the library, in a form described in Clause 17.

3 Annex B recommends lower bounds on the capacity of conforming implementations.

4 Annex C summarizes the evolution of C++ since its first published description, and explains in detail the differences between C++ and C. Certain features of C++ exist solely for compatibility purposes; Annex D describes those features.

5 Throughout this International Standard, each example is introduced by "[*Example:*" and terminated by " — *end example*]". Each note is introduced by "[*Note:*" and terminated by " — *end note*]". Examples and notes may be nested.

1.274 Syntax notation [syntax]

1

In the syntax notation used in this International Standard, syntactic categories are indicated by *italic* type, and literal words and characters in `constant width` type. Alternatives are listed on separate lines except in a few cases where a long set of alternatives is marked by the phrase "one of." If the text of an alternative is too long to fit on a line, the text is continued on subsequent lines indented from the first one. An optional terminal or non-terminal symbol is indicated by the subscript "$_{opt}$", so

`{` *expression*$_{opt}$`}`

indicates an optional expression enclosed in braces.

1 Names for syntactic categories have generally been chosen according to the following rules:

(1.1) — *X-name* is a use of an identifier in a context that determines its meaning (e.g., *class-name*, *typedef-name*).

(1.2) — *X-id* is an identifier with no context-dependent meaning (e.g., *qualified-id*).

(1.3) — *X-seq* is one or more *X*'s without intervening delimiters (e.g., *declaration-seq* is a sequence of declarations).

(1.4) — *X-list* is one or more *X*'s separated by intervening commas (e.g., *expression-list* is a sequence of expressions separated by commas).

3) This documentation also defines implementation-defined behavior; see 1.430.

1.277 The C++ memory model [intro.memory]

1

The fundamental storage unit in the C++ memory model is the *byte*. A byte is at least large enough to contain any member of the basic execution character set (2.154) and the eight-bit code units of the Unicode UTF-8 encoding form and is composed of a contiguous sequence of bits, the number of which is implementation-defined. The least significant bit is called the *low-order bit*; the most significant bit is called the *high-order bit*. The memory available to a C++ program consists of one or more sequences of contiguous bytes. Every byte has a unique address.

1 [*Note:* The representation of types is described in 3.2955. — *end note*]

2

A *memory location* is either an object of scalar type or a maximal sequence of adjacent bit-fields all having non-zero width. [*Note:* Various features of the language, such as references and virtual functions, might involve additional memory locations that are not accessible to programs but are managed by the implementation. — *end note*] Two or more threads of execution (1.859) can update and access separate memory locations without interfering with each other.

1 [*Note:* Thus a bit-field and an adjacent non-bit-field are in separate memory locations, and therefore can be concurrently updated by two threads of execution without interference. The same applies to two bit-fields, if one is declared inside a nested struct declaration and the other is not, or if the two are separated by a zero-length bit-field declaration, or if they are separated by a non-bit-field declaration. It is not safe to concurrently update two bit-fields in the same struct if all fields between them are also bit-fields of non-zero width. — *end note*]

2

[*Example:* A structure declared as

```
struct {
  char a;
  int b:5,
  c:11,
  :0,
  d:8;
  struct {int ee:8;} e;
}
```

contains four separate memory locations: The field `a` and bit-fields `d` and `e.ee` are each separate memory locations, and can be modified concurrently without interfering with each other. The bit-fields `b` and `c` together constitute the fourth memory location. The bit-fields `b` and `c` cannot be concurrently modified, but `b` and `a`, for example, can be. — *end example*]

1.319 The C++ object model [intro.object]

1

The constructs in a C++ program create, destroy, refer to, access, and manipulate objects. An *object* is a region of storage. [*Note:* A function is not an object, regardless of whether or not it occupies storage in the way that objects do. — *end note*] An object is created by a *definition* (3.43), by a *new-expression* (5.2221.1) or by the implementation (12.306) when needed. The properties of an object are determined when the object is created. An object can have a *name* (Clause 3). An object has a *storage duration* (3.2235) which influences its *lifetime* (3.2563). An object has a *type* (3.2955). The term *object type* refers to the type with which the object is created. Some objects are *polymorphic* (10.113); the implementation generates information associated with each such object that makes it possible to determine that object's type during

program execution. For other objects, the interpretation of the values found therein is determined by the type of the *expressions* (Clause 5) used to access them.

1

Objects can contain other objects, called *subobjects*. A subobject can be a *member subobject* (9.194), a *base class subobject* (Clause 10), or an array element. An object that is not a subobject of any other object is called a *complete object*.

1

For every object `x`, there is some object called the *complete object of* `x`, determined as follows:

— If `x` is a complete object, then `x` is the complete object of `x`.

— Otherwise, the complete object of `x` is the complete object of the (unique) object that contains `x`.

1

If a complete object, a data member (9.194), or an array element is of class type, its type is considered the *most derived class*, to distinguish it from the class type of any base class subobject; an object of a most derived class type or of a non-class type is called a *most derived object*.

1

Unless it is a bit-field (9.921), a most derived object shall have a non-zero size and shall occupy one or more bytes of storage. Base class subobjects may have zero size. An object of trivially copyable or standard-layout type (3.2955) shall occupy contiguous bytes of storage.

1

Unless an object is a bit-field or a base class subobject of zero size, the address of that object is the address of the first byte it occupies. Two objects that are not bit-fields may have the same address if one is a subobject of the other, or if at least one is a base class subobject of zero size and they are of different types; otherwise, they shall have distinct addresses.[4]

[*Example:*

```
static const char test1 = 'x';
static const char test2 = 'x';
const bool b = &test1 != &test2;        // always true
```

— *end example*]

1 [*Note:* C++ provides a variety of fundamental types and several ways of composing new types from existing types (3.2955). — *end note*]

1.430 Program execution [intro.execution]

1

The semantic descriptions in this International Standard define a parameterized nondeterministic abstract machine. This International Standard places no requirement on the structure of conforming implementations. In particular, they need not copy or emulate the structure of the abstract machine. Rather, conforming implementations are required to emulate (only) the observable behavior of the abstract machine as explained below.[5]

4) Under the "as-if" rule an implementation is allowed to store two objects at the same machine address or not store an object at all if the program cannot observe the difference (1.430).

5) This provision is sometimes called the "as-if" rule, because an implementation is free to disregard any requirement of this International Standard as long as the result is *as if* the requirement had been obeyed, as far as can be determined from the observable behavior of the program. For instance, an actual implementation need not evaluate part of an expression if it can deduce that its value is not used and that no side effects affecting the observable behavior of the program are produced.

1

Certain aspects and operations of the abstract machine are described in this International Standard as implementation-defined (for example, `sizeof(int)`). These constitute the parameters of the abstract machine. Each implementation shall include documentation describing its characteristics and behavior in these respects.[6] Such documentation shall define the instance of the abstract machine that corresponds to that implementation (referred to as the "corresponding instance" below).

1

Certain other aspects and operations of the abstract machine are described in this International Standard as unspecified (for example, evaluation of expressions in a *new-initializer* if the allocation function fails to allocate memory (5.2221.1)). Where possible, this International Standard defines a set of allowable behaviors. These define the nondeterministic aspects of the abstract machine. An instance of the abstract machine can thus have more than one possible execution for a given program and a given input.

1

Certain other operations are described in this International Standard as undefined (for example, the effect of attempting to modify a `const` object). [*Note:* This International Standard imposes no requirements on the behavior of programs that contain undefined behavior. — *end note*]

1

A conforming implementation executing a well-formed program shall produce the same observable behavior as one of the possible executions of the corresponding instance of the abstract machine with the same program and the same input. However, if any such execution contains an undefined operation, this International Standard places no requirement on the implementation executing that program with that input (not even with regard to operations preceding the first undefined operation).

1

If a signal handler is executed as a result of a call to the `raise` function, then the execution of the handler is sequenced after the invocation of the `raise` function and before its return. [*Note:* When a signal is received for another reason, the execution of the signal handler is usually unsequenced with respect to the rest of the program. — *end note*]

1

An instance of each object with automatic storage duration (3.2301.1) is associated with each entry into its block. Such an object exists and retains its last-stored value during the execution of the block and while the block is suspended (by a call of a function or receipt of a signal).

1

The least requirements on a conforming implementation are:

— Access to volatile objects are evaluated strictly according to the rules of the abstract machine.

— At program termination, all data written into files shall be identical to one of the possible results that execution of the program according to the abstract semantics would have produced.

— The input and output dynamics of interactive devices shall take place in such a fashion that prompting output is actually delivered before a program waits for input. What constitutes an interactive device is implementation-defined.

These collectively are referred to as the *observable behavior* of the program. [*Note:* More stringent correspondences between abstract and actual semantics may be defined by each implementation. — *end note*]

1

6) This documentation also includes conditionally-supported constructs and locale-specific behavior. See 1.196.

[*Note:* Operators can be regrouped according to the usual mathematical rules only where the operators really are associative or commutative.[7] For example, in the following fragment

```
int a, b;
/* ... */
a = a + 32760 + b + 5;
```

the expression statement behaves exactly the same as

```
a = (((a + 32760) + b) + 5);
```

due to the associativity and precedence of these operators. Thus, the result of the sum `(a + 32760)` is next added to `b`, and that result is then added to 5 which results in the value assigned to `a`. On a machine in which overflows produce an exception and in which the range of values representable by an `int` is `[-32768,+32767]`, the implementation cannot rewrite this expression as

```
a = ((a + b) + 32765);
```

since if the values for `a` and `b` were, respectively, -32754 and -15, the sum `a + b` would produce an exception while the original expression would not; nor can the expression be rewritten either as

```
a = ((a + 32765) + b);
```

or

```
a = (a + (b + 32765));
```

since the values for `a` and `b` might have been, respectively, 4 and -8 or -17 and 12. However on a machine in which overflows do not produce an exception and in which the results of overflows are reversible, the above expression statement can be rewritten by the implementation in any of the above ways because the same result will occur. — *end note*]

1

A *full-expression* is an expression that is not a subexpression of another expression. [*Note:* in some contexts, such as unevaluated operands, a syntactic subexpression is considered a full-expression (Clause 5). — *end note*] If a language construct is defined to produce an implicit call of a function, a use of the language construct is considered to be an expression for the purposes of this definition. A call to a destructor generated at the end of the lifetime of an object other than a temporary object is an implicit full-expression. Conversions applied to the result of an expression in order to satisfy the requirements of the language construct in which the expression appears are also considered to be part of the full-expression.

[*Example:*

```
struct S {
  S(int i): I(i) { }
  int& v() { return I; }
private:
  int I;
};

 S s1(1);            // full-expression is call of S::S(int)
 S s2 = 2;           // full-expression is call of S::S(int)

void f() {
  if (S(3).v())      // full-expression includes lvalue-to-rvalue and
                     // int to bool conversions, performed before
```

7) Overloaded operators are never assumed to be associative or commutative.

```
                    // temporary is deleted at end of full-expression
  { }
}
```

— *end example*]

1

[*Note:* The evaluation of a full-expression can include the evaluation of subexpressions that are not lexically part of the full-expression. For example, subexpressions involved in evaluating default arguments (8.545.1) are considered to be created in the expression that calls the function, not the expression that defines the default argument. — *end note*]

1

Accessing an object designated by a `volatile` glvalue (3.3673), modifying an object, calling a library I/O function, or calling a function that does any of those operations are all *side effects*, which are changes in the state of the execution environment. *Evaluation* of an expression (or a sub-expression) in general includes both value computations (including determining the identity of an object for glvalue evaluation and fetching a value previously assigned to an object for prvalue evaluation) and initiation of side effects. When a call to a library I/O function returns or an access to a `volatile` object is evaluated the side effect is considered complete, even though some external actions implied by the call (such as the I/O itself) or by the `volatile` access may not have completed yet.

1

Sequenced before is an asymmetric, transitive, pair-wise relation between evaluations executed by a single thread (1.859), which induces a partial order among those evaluations. Given any two evaluations A and B, if A is sequenced before B, then the execution of A shall precede the execution of B. If A is not sequenced before B and B is not sequenced before A, then A and B are *unsequenced*. [*Note:* The execution of unsequenced evaluations can overlap. — *end note*] Evaluations A and B are *indeterminately sequenced* when either A is sequenced before B or B is sequenced before A, but it is unspecified which. [*Note:* Indeterminately sequenced evaluations cannot overlap, but either could be executed first. — *end note*]

1

Every value computation and side effect associated with a full-expression is sequenced before every value computation and side effect associated with the next full-expression to be evaluated.[8]

1

Except where noted, evaluations of operands of individual operators and of subexpressions of individual expressions are unsequenced. [*Note:* In an expression that is evaluated more than once during the execution of a program, unsequenced and indeterminately sequenced evaluations of its subexpressions need not be performed consistently in different evaluations. — *end note*] The value computations of the operands of an operator are sequenced before the value computation of the result of the operator. If a side effect on a scalar object is unsequenced relative to either another side effect on the same scalar object or a value computation using the value of the same scalar object, and they are not potentially concurrent (1.859), the behavior is undefined. [*Note:* The next section imposes similar, but more complex restrictions on potentially concurrent computations. — *end note*]

[*Example:*

```
void f(int, int);
void g(int i, int* v) {
```

8) As specified in 12.306, after a full-expression is evaluated, a sequence of zero or more invocations of destructor functions for temporary objects takes place, usually in reverse order of the construction of each temporary object.

```
  i = v[i++];          // the behavior is undefined
  i = 7, i++, i++;     // i becomes 9

  i = i++ + 1;         // the behavior is undefined
  i = i + 1;           // the value of i is incremented

  f(i = -1, i = -1);   // the behavior is undefined
}
```

— *end example*]

When calling a function (whether or not the function is inline), every value computation and side effect associated with any argument expression, or with the postfix expression designating the called function, is sequenced before execution of every expression or statement in the body of the called function. [*Note:* Value computations and side effects associated with different argument expressions are unsequenced. — *end note*] Every evaluation in the calling function (including other function calls) that is not otherwise specifically sequenced before or after the execution of the body of the called function is indeterminately sequenced with respect to the execution of the called function.[9] Several contexts in C++ cause evaluation of a function call, even though no corresponding function call syntax appears in the translation unit. [*Example:* Evaluation of a `new` expression invokes one or more allocation and constructor functions; see 5.2221.1. For another example, invocation of a conversion function (12.742.1) can arise in contexts in which no function call syntax appears. — *end example*] The sequencing constraints on the execution of the called function (as described above) are features of the function calls as evaluated, whatever the syntax of the expression that calls the function might be.

1.859 Multi-threaded executions and data races [intro.multithread]

1

A *thread of execution* (also known as a *thread*) is a single flow of control within a program, including the initial invocation of a specific top-level function, and recursively including every function invocation subsequently executed by the thread. [*Note:* When one thread creates another, the initial call to the top-level function of the new thread is executed by the new thread, not by the creating thread. — *end note*] Every thread in a program can potentially access every object and function in a program.[10] Under a hosted implementation, a C++ program can have more than one thread running concurrently. The execution of each thread proceeds as defined by the remainder of this standard. The execution of the entire program consists of an execution of all of its threads. [*Note:* Usually the execution can be viewed as an interleaving of all its threads. However, some kinds of atomic operations, for example, allow executions inconsistent with a simple interleaving, as described below. — *end note*] Under a freestanding implementation, it is implementation-defined whether a program can have more than one thread of execution.

1

A signal handler that is executed as a result of a call to the `raise` function belongs to the same thread of execution as the call to the `raise` function. Otherwise it is unspecified which thread of execution contains a signal handler invocation.

1

Implementations should ensure that all unblocked threads eventually make progress. [*Note:* Standard library functions may silently block on I/O or locks. Factors in the execution environment, including externally-imposed thread priorities, may prevent an implementation from making certain guarantees of forward progress. — *end note*]

9) In other words, function executions do not interleave with each other.

10) An object with automatic or thread storage duration (3.2235) is associated with one specific thread, and can be accessed by a different thread only indirectly through a pointer or reference (3.3568.1).

1

Executions of atomic functions that are either defined to be lock-free (29.77) or indicated as lock-free (29.31) are *lock-free executions*.

— If there is only one unblocked thread, a lock-free execution in that thread shall complete. [*Note:* Concurrently executing threads may prevent progress of a lock-free execution. For example, this situation can occur with load-locked store-conditional implementations. This property is sometimes termed obstruction-free. — *end note*]

— When one or more lock-free executions run concurrently, at least one should complete. [*Note:* It is difficult for some implementations to provide absolute guarantees to this effect, since repeated and particularly inopportune interference from other threads may prevent forward progress, e.g., by repeatedly stealing a cache line for unrelated purposes between load-locked and store-conditional instructions. Implementations should ensure that such effects cannot indefinitely delay progress under expected operating conditions, and that such anomalies can therefore safely be ignored by programmers. Outside this International Standard, this property is sometimes termed lock-free. — *end note*]

1

The value of an object visible to a thread T at a particular point is the initial value of the object, a value assigned to the object by T, or a value assigned to the object by another thread, according to the rules below. [*Note:* In some cases, there may instead be undefined behavior. Much of this section is motivated by the desire to support atomic operations with explicit and detailed visibility constraints. However, it also implicitly supports a simpler view for more restricted programs. — *end note*]

1

Two expression evaluations *conflict* if one of them modifies a memory location (1.277) and the other one accesses or modifies the same memory location.

1

The library defines a number of atomic operations (Clause 29) and operations on mutexes (Clause 30) that are specially identified as synchronization operations. These operations play a special role in making assignments in one thread visible to another. A synchronization operation on one or more memory locations is either a consume operation, an acquire operation, a release operation, or both an acquire and release operation. A synchronization operation without an associated memory location is a fence and can be either an acquire fence, a release fence, or both an acquire and release fence. In addition, there are relaxed atomic operations, which are not synchronization operations, and atomic read-modify-write operations, which have special characteristics. [*Note:* For example, a call that acquires a mutex will perform an acquire operation on the locations comprising the mutex. Correspondingly, a call that releases the same mutex will perform a release operation on those same locations. Informally, performing a release operation on A forces prior side effects on other memory locations to become visible to other threads that later perform a consume or an acquire operation on A. "Relaxed" atomic operations are not synchronization operations even though, like synchronization operations, they cannot contribute to data races. — *end note*]

1

All modifications to a particular atomic object M occur in some particular total order, called the *modification order* of M. If A and B are modifications of an atomic object M and A happens before (as defined below) B, then A shall precede B in the modification order of M, which is defined below. [*Note:* This states that the modification orders must respect the "happens before" relationship. — *end note*] [*Note:* There is a separate order for each atomic object. There is no requirement that these can be combined into a single total order for all objects. In general this will be impossible since different threads may observe modifications to different objects in inconsistent orders. — *end note*]

1

A *release sequence* headed by a release operation A on an atomic object M is a maximal contiguous subsequence of side effects in the modification order of M, where the first operation is `A`, and every subsequent operation

— is performed by the same thread that performed `A`, or

— is an atomic read-modify-write operation.

1

Certain library calls *synchronize with* other library calls performed by another thread. For example, an atomic store-release synchronizes with a load-acquire that takes its value from the store (29.3). [*Note:* Except in the specified cases, reading a later value does not necessarily ensure visibility as described below. Such a requirement would sometimes interfere with efficient implementation. — *end note*] [*Note:* The specifications of the synchronization operations define when one reads the value written by another. For atomic objects, the definition is clear. All operations on a given mutex occur in a single total order. Each mutex acquisition "reads the value written" by the last mutex release. — *end note*]

1

An evaluation A *carries a dependency* to an evaluation B if

— the value of A is used as an operand of B, unless:

  — B is an invocation of any specialization of `std::kill_dependency` (29.3), or

  — A is the left operand of a built-in logical AND (`&&`, see 5.3286) or logical OR (`||`, see 5.3322) operator, or

  — A is the left operand of a conditional (`?:`, see 5.3337) operator, or

  — A is the left operand of the built-in comma (`,`) operator (5.3602);

  or

— A writes a scalar object or bit-field M, B reads the value written by A from M, and A is sequenced before B, or

— for some evaluation X, A carries a dependency to X, and X carries a dependency to B.

[*Note:* "Carries a dependency to" is a subset of "is sequenced before", and is similarly strictly intra-thread. — *end note*]

1

An evaluation A is *dependency-ordered before* an evaluation B if

— A performs a release operation on an atomic object M, and, in another thread, B performs a consume operation on M and reads a value written by any side effect in the release sequence headed by A, or

— for some evaluation X, A is dependency-ordered before X and X carries a dependency to B.

[*Note:* The relation "is dependency-ordered before" is analogous to "synchronizes with", but uses release/-consume in place of release/acquire. — *end note*]

1

An evaluation A *inter-thread happens before* an evaluation B if

— A synchronizes with B, or

— A is dependency-ordered before B, or

— for some evaluation X
  — A synchronizes with X and X is sequenced before B, or
  — A is sequenced before X and X inter-thread happens before B, or
  — A inter-thread happens before X and X inter-thread happens before B.

[*Note:* The "inter-thread happens before" relation describes arbitrary concatenations of "sequenced before", "synchronizes with" and "dependency-ordered before" relationships, with two exceptions. The first exception is that a concatenation is not permitted to end with "dependency-ordered before" followed by "sequenced before". The reason for this limitation is that a consume operation participating in a "dependency-ordered before" relationship provides ordering only with respect to operations to which this consume operation actually carries a dependency. The reason that this limitation applies only to the end of such a concatenation is that any subsequent release operation will provide the required ordering for a prior consume operation. The second exception is that a concatenation is not permitted to consist entirely of "sequenced before". The reasons for this limitation are (1) to permit "inter-thread happens before" to be transitively closed and (2) the "happens before" relation, defined below, provides for relationships consisting entirely of "sequenced before". — *end note*]

1 An evaluation A *happens before* an evaluation B if:

(1.1) — A is sequenced before B, or

(1.2) — A inter-thread happens before B.

The implementation shall ensure that no program execution demonstrates a cycle in the "happens before" relation. [*Note:* This cycle would otherwise be possible only through the use of consume operations. — *end note*]

2

A *visible side effect* A on a scalar object or bit-field M with respect to a value computation B of M satisfies the conditions:

— A happens before B and

— there is no other side effect X to M such that A happens before X and X happens before B.

The value of a non-atomic scalar object or bit-field M, as determined by evaluation B, shall be the value stored by the visible side effect A. [*Note:* If there is ambiguity about which side effect to a non-atomic object or bit-field is visible, then the behavior is either unspecified or undefined. — *end note*] [*Note:* This states that operations on ordinary objects are not visibly reordered. This is not actually detectable without data races, but it is necessary to ensure that data races, as defined below, and with suitable restrictions on the use of atomics, correspond to data races in a simple interleaved (sequentially consistent) execution. — *end note*]

1 The value of an atomic object M, as determined by evaluation B, shall be the value stored by some side effect A that modifies M, where B does not happen before A. [*Note:* The set of such side effects is also restricted by the rest of the rules described here, and in particular, by the coherence requirements below. — *end note*]

2 If an operation A that modifies an atomic object M happens before an operation B that modifies M, then A shall be earlier than B in the modification order of M. [*Note:* This requirement is known as write-write coherence. — *end note*]

3

If a value computation A of an atomic object M happens before a value computation B of M, and A takes its value from a side effect X on M, then the value computed by B shall either be the value stored by X or

the value stored by a side effect Y on M, where Y follows X in the modification order of M. [*Note:* This requirement is known as read-read coherence. — *end note*]

1 If a value computation A of an atomic object M happens before an operation B that modifies M, then A shall take its value from a side effect X on M, where X precedes B in the modification order of M. [*Note:* This requirement is known as read-write coherence. — *end note*]

2 If a side effect X on an atomic object M happens before a value computation B of M, then the evaluation B shall take its value from X or from a side effect Y that follows X in the modification order of M. [*Note:* This requirement is known as write-read coherence. — *end note*]

1 [*Note:* The four preceding coherence requirements effectively disallow compiler reordering of atomic operations to a single object, even if both operations are relaxed loads. This effectively makes the cache coherence guarantee provided by most hardware available to C++ atomic operations. — *end note*]

1 [*Note:* The value observed by a load of an atomic depends on the "happens before" relation, which depends on the values observed by loads of atomics. The intended reading is that there must exist an association of atomic loads with modifications they observe that, together with suitably chosen modification orders and the "happens before" relation derived as described above, satisfy the resulting constraints as imposed here. — *end note*]

2 Two actions are *potentially concurrent* if

— they are performed by different threads, or

— they are unsequenced, and at least one is performed by a signal handler.

The execution of a program contains a *data race* if it contains two potentially concurrent conflicting actions, at least one of which is not atomic, and neither happens before the other, except for the special case for signal handlers described below. Any such data race results in undefined behavior. [*Note:* It can be shown that programs that correctly use mutexes and `memory_order_seq_cst` operations to prevent all data races and use no other synchronization operations behave as if the operations executed by their constituent threads were simply interleaved, with each value computation of an object being taken from the last side effect on that object in that interleaving. This is normally referred to as "sequential consistency". However, this applies only to data-race-free programs, and data-race-free programs cannot observe most program transformations that do not change single-threaded program semantics. In fact, most single-threaded program transformations continue to be allowed, since any program that behaves differently as a result must perform an undefined operation. — *end note*]

1 Two accesses to the same object of type `volatile sig_atomic_t` do not result in a data race if both occur in the same thread, even if one or more occurs in a signal handler. For each signal handler invocation, evaluations performed by the thread invoking a signal handler can be divided into two groups A and B, such that no evaluations in B happen before evaluations in A, and the evaluations of such `volatile sig_atomic_t` objects take values as though all evaluations in A happened before the execution of the signal handler and the execution of the signal handler happened before all evaluations in B.

2 [*Note:* Compiler transformations that introduce assignments to a potentially shared memory location that would not be modified by the abstract machine are generally precluded by this standard, since such an assignment might overwrite another assignment by a different thread in cases in which an abstract machine execution would not have encountered a data race. This includes implementations of data member assign-

ment that overwrite adjacent members in separate memory locations. Reordering of atomic loads in cases in which the atomics in question may alias is also generally precluded, since this may violate the coherence rules. — *end note*]

1

[*Note:* Transformations that introduce a speculative read of a potentially shared memory location may not preserve the semantics of the C++ program as defined in this standard, since they potentially introduce a data race. However, they are typically valid in the context of an optimizing compiler that targets a specific machine with well-defined semantics for data races. They would be invalid for a hypothetical machine that is not tolerant of races or provides hardware race detection. — *end note*]

1

The implementation may assume that any thread will eventually do one of the following:

— terminate,

— make a call to a library I/O function,

— access or modify a volatile object, or

— perform a synchronization operation or an atomic operation.

[*Note:* This is intended to allow compiler transformations such as removal of empty loops, even when termination cannot be proven. — *end note*]

1

An implementation should ensure that the last value (in modification order) assigned by an atomic or synchronization operation will become visible to all other threads in a finite period of time.

1.1005 Acknowledgments [intro.ack]

1 The C++ programming language as described in this International Standard is based on the language as described in Chapter R (Reference Manual) of Stroustrup: *The C++ Programming Language* (second edition, Addison-Wesley Publishing Company, ISBN 0-201-53992-6, copyright ©1991 AT&T). That, in turn, is based on the C programming language as described in Appendix A of Kernighan and Ritchie: *The C Programming Language* (Prentice-Hall, 1978, ISBN 0-13-110163-3, copyright ©1978 AT&T).

2 Portions of the library Clauses of this International Standard are based on work by P.J. Plauger, which was published as *The Draft Standard C++ Library* (Prentice-Hall, ISBN 0-13-117003-1, copyright ©1995 P.J. Plauger).

3 POSIX® is a registered trademark of the Institute of Electrical and Electronic Engineers, Inc.

4 All rights in these originals are reserved.

2 Lexical conventions [lex]

2.1 Separate translation [lex.separate]

1

The text of the program is kept in units called *source files* in this International Standard. A source file together with all the headers (17.48.0.1) and source files included (16.48) via the preprocessing directive `#include`, less any source lines skipped by any of the conditional inclusion (16.18) preprocessing directives, is called a *translation unit*. [*Note:* A C++ program need not all be translated at the same time. — *end note*]

1 [*Note:* Previously translated translation units and instantiation units can be preserved individually or in libraries. The separate translation units of a program communicate (3.1459) by (for example) calls to functions whose identifiers have external linkage, manipulation of objects whose identifiers have external linkage, or manipulation of data files. Translation units can be separately translated and then later linked to produce an executable program (3.1459). — *end note*]

2.6 Phases of translation [lex.phases]

1

The precedence among the syntax rules of translation is specified by the following phases.[11]

1. Physical source file characters are mapped, in an implementation-defined manner, to the basic source character set (introducing new-line characters for end-of-line indicators) if necessary. The set of physical source file characters accepted is implementation-defined. Trigraph sequences (2.211) are replaced by corresponding single-character internal representations. Any source file character not in the basic source character set (2.154) is replaced by the universal-character-name that designates that character. (An implementation may use any internal encoding, so long as an actual extended character encountered in the source file, and the same extended character expressed in the source file as a universal-character-name (i.e., using the `\uXXXX` notation), are handled equivalently except where this replacement is reverted in a raw string literal.)

2. Each instance of a backslash character (\) immediately followed by a new-line character is deleted, splicing physical source lines to form logical source lines. Only the last backslash on any physical source line shall be eligible for being part of such a splice. Except for splices reverted in a raw string literal, if a splice results in a character sequence that matches the syntax of a universal-character-name, the behavior is undefined. A source file that is not empty and that does not end in a new-line character, or that ends in a new-line character immediately preceded by a backslash character before any such splicing takes place, shall be processed as if an additional new-line character were appended to the file.

3. The source file is decomposed into preprocessing tokens (2.214) and sequences of white-space characters (including comments). A source file shall not end in a partial preprocessing token or in a partial comment.[12] Each comment is replaced by one space character. New-line characters are retained. Whether each nonempty sequence of white-space characters other than new-line is retained or replaced by one

11) Implementations must behave as if these separate phases occur, although in practice different phases might be folded together.

12) A partial preprocessing token would arise from a source file ending in the first portion of a multi-character token that requires a terminating sequence of characters, such as a *header-name* that is missing the closing `"` or `>`. A partial comment would arise from a source file ending with an unclosed `/*` comment.

space character is unspecified. The process of dividing a source file's characters into preprocessing tokens is context-dependent. [*Example:* see the handling of `<` within a `#include` preprocessing directive. — *end example*]

4. Preprocessing directives are executed, macro invocations are expanded, and `_Pragma` unary operator expressions are executed. If a character sequence that matches the syntax of a universal-character-name is produced by token concatenation (16.175.1), the behavior is undefined. A `#include` preprocessing directive causes the named header or source file to be processed from phase 1 through phase 4, recursively. All preprocessing directives are then deleted.

5. Each source character set member in a character literal or a string literal, as well as each escape sequence and universal-character-name in a character literal or a non-raw string literal, is converted to the corresponding member of the execution character set (2.356.1, 2.453.1); if there is no corresponding member, it is converted to an implementation-defined member other than the null (wide) character.[13]

6. Adjacent string literal tokens are concatenated.

7. White-space characters separating tokens are no longer significant. Each preprocessing token is converted into a token. (2.247). The resulting tokens are syntactically and semantically analyzed and translated as a translation unit. [*Note:* The process of analyzing and translating the tokens may occasionally result in one token being replaced by a sequence of other tokens (14.224). — *end note*] [*Note:* Source files, translation units and translated translation units need not necessarily be stored as files, nor need there be any one-to-one correspondence between these entities and any external representation. The description is conceptual only, and does not specify any particular implementation. — *end note*]

8. Translated translation units and instantiation units are combined as follows: [*Note:* Some or all of these may be supplied from a library. — *end note*] Each translated translation unit is examined to produce a list of required instantiations. [*Note:* This may include instantiations which have been explicitly requested (14.1841.1). — *end note*] The definitions of the required templates are located. It is implementation-defined whether the source of the translation units containing these definitions is required to be available. [*Note:* An implementation could encode sufficient information into the translated translation unit so as to ensure the source is not required here. — *end note*] All the required instantiations are performed to produce *instantiation units*. [*Note:* These are similar to translated translation units, but contain no references to uninstantiated templates and no template definitions. — *end note*] The program is ill-formed if any instantiation fails.

9. All external entity references are resolved. Library components are linked to satisfy external references to entities not defined in the current translation. All such translator output is collected into a program image which contains information needed for execution in its execution environment.

2.154 Character sets [lex.charset]

1

The *basic source character set* consists of 96 characters: the space character, the control characters representing horizontal tab, vertical tab, form feed, and new-line, plus the following 91 graphical characters:[14]

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

13) An implementation need not convert all non-corresponding source characters to the same execution character.

14) The glyphs for the members of the basic source character set are intended to identify characters from the subset of ISO/IEC 10646 which corresponds to the ASCII character set. However, because the mapping from source file characters to the source character set (described in translation phase 1) is specified as implementation-defined, an implementation is required to document how the basic source characters are represented in source files.

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

0 1 2 3 4 5 6 7 8 9

_ { } [ ] # ( ) < > % : ; . ? * + - / ^ & | ~ ! = , \ " '
```

1

The *universal-character-name* construct provides a way to name other characters.

hex-quad:
hexadecimal-digit hexadecimal-digit hexadecimal-digit hexadecimal-digit

universal-character-name:
`\u` *hex-quad*
`\U` *hex-quad hex-quad*

The character designated by the universal-character-name `\UNNNNNNNN` is that character whose character short name in ISO/IEC 10646 is `NNNNNNNN`; the character designated by the universal-character-name `\uNNNN` is that character whose character short name in ISO/IEC 10646 is `0000NNNN`. If the hexadecimal value for a universal-character-name corresponds to a surrogate code point (in the range 0xD800–0xDFFF, inclusive), the program is ill-formed. Additionally, if the hexadecimal value for a universal-character-name outside the *c-char-sequence*, *s-char-sequence*, or *r-char-sequence* of a character or string literal corresponds to a control character (in either of the ranges 0x00–0x1F or 0x7F–0x9F, both inclusive) or to a character in the basic source character set, the program is ill-formed.[15]

1

The *basic execution character set* and the *basic execution wide-character set* shall each contain all the members of the basic source character set, plus control characters representing alert, backspace, and carriage return, plus a *null character* (respectively, *null wide character*), whose representation has all zero bits. For each basic execution character set, the values of the members shall be non-negative and distinct from one another. In both the source and execution basic character sets, the value of each character after `0` in the above list of decimal digits shall be one greater than the value of the previous. The *execution character set* and the *execution wide-character set* are implementation-defined supersets of the basic execution character set and the basic execution wide-character set, respectively. The values of the members of the execution character sets and the sets of additional members are locale-specific.

2.211 Trigraph sequences [lex.trigraph]

1

Before any other processing takes place, each occurrence of one of the following sequences of three characters ("*trigraph sequences*") is replaced by the single character indicated in Table 1.

Table 1 — Trigraph sequences

Trigraph	Replacement	Trigraph	Replacement	Trigraph	Replacement
`??=`	`#`	`??(`	`[`	`??<`	`{`
`??/`	`\`	`??)`	`]`	`??>`	`}`
`??'`	`^`	`??!`	`|`	`??-`	`~`

1 [*Example:*

```
??=define arraycheck(a,b) a??(b??) ??!??! b??(a??)
```

15) A sequence of characters resembling a universal-character-name in an *r-char-sequence* (2.453.1) does not form a universal-character-name.

becomes

```
#define arraycheck(a,b) a[b] || b[a]
```

— *end example*]

2 No other trigraph sequence exists. Each `?` that does not begin one of the trigraphs listed above is not changed.

2.214 Preprocessing tokens [lex.pptoken]

preprocessing-token:
- *header-name*
- *identifier*
- *pp-number*
- *character-literal*
- *user-defined-character-literal*
- *string-literal*
- *user-defined-string-literal*
- *preprocessing-op-or-punc*
- each non-white-space character that cannot be one of the above

1 Each preprocessing token that is converted to a token (2.247) shall have the lexical form of a keyword, an identifier, a literal, an operator, or a punctuator.

2

A preprocessing token is the minimal lexical element of the language in translation phases 3 through 6. The categories of preprocessing token are: header names, identifiers, preprocessing numbers, character literals (including user-defined character literals), string literals (including user-defined string literals), preprocessing operators and punctuators, and single non-white-space characters that do not lexically match the other preprocessing token categories. If a `'` or a `"` character matches the last category, the behavior is undefined. Preprocessing tokens can be separated by white space; this consists of comments (2.252), or white-space characters (space, horizontal tab, new-line, vertical tab, and form-feed), or both. As described in Clause 16, in certain circumstances during translation phase 4, white space (or the absence thereof) serves as more than preprocessing token separation. White space can appear within a preprocessing token only as part of a header name or between the quotation characters in a character literal or string literal.

1

If the input stream has been parsed into preprocessing tokens up to a given character:

— If the next character begins a sequence of characters that could be the prefix and initial double quote of a raw string literal, such as `R"`, the next preprocessing token shall be a raw string literal. Between the initial and final double quote characters of the raw string, any transformations performed in phases 1 and 2 (trigraphs, universal-character-names, and line splicing) are reverted; this reversion shall apply before any *d-char*, *r-char*, or delimiting parenthesis is identified. The raw string literal is defined as the shortest sequence of characters that matches the raw-string pattern

  encoding-prefix$_{opt}$`R` *raw-string*

— Otherwise, if the next three characters are `<::` and the subsequent character is neither `:` nor `>`, the `<` is treated as a preprocessor token by itself and not as the first character of the alternative token `<:`.

— Otherwise, the next preprocessing token is the longest sequence of characters that could constitute a preprocessing token, even if that would cause further lexical analysis to fail.

[*Example:*

```
#define R "x"
const char* s = R"y";           // ill-formed raw string, not "x" "y"
```

— *end example*]

1 [*Example:* The program fragment `1Ex` is parsed as a preprocessing number token (one that is not a valid floating or integer literal token), even though a parse as the pair of preprocessing tokens `1` and `Ex` might produce a valid expression (for example, if `Ex` were a macro defined as `+1`). Similarly, the program fragment `1E1` is parsed as a preprocessing number (one that is a valid floating literal token), whether or not `E` is a macro name. — *end example*]

2 [*Example:* The program fragment `x+++++y` is parsed as `x ++ ++ + y`, which, if `x` and `y` have integral types, violates a constraint on increment operators, even though the parse `x ++ + ++ y` might yield a correct expression. — *end example*]

2.236 Alternative tokens [lex.digraph]

1

Alternative token representations are provided for some operators and punctuators.[16]

1

In all respects of the language, each alternative token behaves the same, respectively, as its primary token, except for its spelling.[17] The set of alternative tokens is defined in Table 2.

Table 2 — Alternative tokens

Alternative	Primary	Alternative	Primary	Alternative	Primary
`<%`	`{`	`and`	`&&`	`and_eq`	`&=`
`%>`	`}`	`bitor`	`|`	`or_eq`	`|=`
`<:`	`[`	`or`	`||`	`xor_eq`	`^=`
`:>`	`]`	`xor`	`^`	`not`	`!`
`%:`	`#`	`compl`	`~`	`not_eq`	`!=`
`%:%:`	`##`	`bitand`	`&`		

2.247 Tokens [lex.token]

token:
- *identifier*
- *keyword*
- *literal*
- *operator*
- *punctuator*

1

There are five kinds of tokens: identifiers, keywords, literals,[18] operators, and other separators. Blanks, horizontal and vertical tabs, newlines, formfeeds, and comments (collectively, "white space"), as described below, are ignored except as they serve to separate tokens. [*Note:* Some white space is required to separate otherwise adjacent identifiers, keywords, numeric literals, and alternative tokens containing alphabetic characters. — *end note*]

2.252 Comments [lex.comment]

1

16) These include "digraphs" and additional reserved words. The term "digraph" (token consisting of two characters) is not perfectly descriptive, since one of the alternative preprocessing-tokens is `%:%:` and of course several primary tokens contain two characters. Nonetheless, those alternative tokens that aren't lexical keywords are colloquially known as "digraphs".

17) Thus the "stringized" values (16.161.1) of `[` and `<:` will be different, maintaining the source spelling, but the tokens can otherwise be freely interchanged.

18) Literals include strings and character and numeric literals.

The characters `/*` start a comment, which terminates with the characters `*/`. These comments do not nest. The characters `//` start a comment, which terminates immediately before the next new-line character. If there is a form-feed or a vertical-tab character in such a comment, only white-space characters shall appear between it and the new-line that terminates the comment; no diagnostic is required. [*Note:* The comment characters `//`, `/*`, and `*/` have no special meaning within a `//` comment and are treated just like other characters. Similarly, the comment characters `//` and `/*` have no special meaning within a `/*` comment. — *end note*]

2.257 Header names [lex.header]

header-name:
  `<` *h-char-sequence* `>`
  `"` *q-char-sequence* `"`

h-char-sequence:
  h-char
  h-char-sequence h-char

h-char:
  any member of the source character set except new-line and `>`

q-char-sequence:
  q-char
  q-char-sequence q-char

q-char:
  any member of the source character set except new-line and `"`

1

Header name preprocessing tokens shall only appear within a `#include` preprocessing directive (16.48). The sequences in both forms of *header-name*s are mapped in an implementation-defined manner to headers or to external source file names as specified in 16.48.

1

The appearance of either of the characters `'` or `\` or of either of the character sequences `/*` or `//` in a *q-char-sequence* or an *h-char-sequence* is conditionally-supported with implementation-defined semantics, as is the appearance of the character `"` in an *h-char-sequence*.[19]

2.266 Preprocessing numbers [lex.ppnumber]

pp-number:
  digit
  `.` *digit*
  pp-number digit
  pp-number identifier-nondigit
  pp-number `'` *digit*
  pp-number `'` *nondigit*
  pp-number `e` *sign*
  pp-number `E` *sign*
  pp-number `.`

1 Preprocessing number tokens lexically include all integer literal tokens (2.300.1) and all floating literal tokens (2.431.1).

2

A preprocessing number does not have a type or a value; it acquires both after a successful conversion to an integer literal token or a floating literal token.

19) Thus, a sequence of characters that resembles an escape sequence might result in an error, be interpreted as the character corresponding to the escape sequence, or have a completely different meaning, depending on the implementation.

2.268 Identifiers [lex.name]

identifier:
- *identifier-nondigit*
- *identifier identifier-nondigit*
- *identifier digit*

identifier-nondigit:
- *nondigit*
- *universal-character-name*
- other implementation-defined characters

nondigit: one of

```
a b c d e f g h i j k l m
n o p q r s t u v w x y z
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z _
```

digit: one of

```
0 1 2 3 4 5 6 7 8 9
```

1

An identifier is an arbitrarily long sequence of letters and digits. Each universal-character-name in an identifier shall designate a character whose encoding in ISO 10646 falls into one of the ranges specified in E.1. The initial element shall not be a universal-character-name designating a character whose encoding falls into one of the ranges specified in E.2. Upper- and lower-case letters are different. All characters are significant.[20]

1

The identifiers in Table 3 have a special meaning when appearing in a certain context. When referred to in the grammar, these identifiers are used explicitly rather than using the *identifier* grammar production. Unless otherwise specified, any ambiguity as to whether a given *identifier* has a special meaning is resolved to interpret the token as a regular *identifier*.

Table 3 — Identifiers with special meaning

`override`	`final`

1

In addition, some identifiers are reserved for use by C++ implementations and standard libraries (17.332.0.0.1) and shall not be used otherwise; no diagnostic is required.

2.287 Keywords [lex.key]

1

The identifiers shown in Table 4 are reserved for use as keywords (that is, they are unconditionally treated as keywords in phase 7) except in an *attribute-token* (7.1546.1) [*Note:* The `export` keyword is unused but is reserved for future use. — *end note*]:

1

Furthermore, the alternative representations shown in Table 5 for certain operators and punctuators (2.236) are reserved and shall not be used otherwise:

20) On systems in which linkers cannot accept extended characters, an encoding of the universal-character-name may be used in forming valid external identifiers. For example, some otherwise unused character or sequence of characters may be used to encode the `\u` in a universal-character-name. Extended characters may produce a long external identifier, but C++ does not place a translation limit on significant characters for external identifiers. In C++, upper- and lower-case letters are considered different for all identifiers, including external identifiers.

Table 4 — Keywords

alignas	continue	friend	register	true
alignof	decltype	goto	reinterpret_cast	try
asm	default	if	return	typedef
auto	delete	inline	short	typeid
bool	do	int	signed	typename
break	double	long	sizeof	union
case	dynamic_cast	mutable	static	unsigned
catch	else	namespace	static_assert	using
char	enum	new	static_cast	virtual
char16_t	explicit	noexcept	struct	void
char32_t	export	nullptr	switch	volatile
class	extern	operator	template	wchar_t
const	false	private	this	while
constexpr	float	protected	thread_local	
const_cast	for	public	throw	

Table 5 — Alternative representations

and	and_eq	bitand	bitor	compl	not
not_eq	or	or_eq	xor	xor_eq	

2.294 Operators and punctuators [lex.operators]

1

The lexical representation of C++ programs includes a number of preprocessing tokens which are used in the syntax of the preprocessor or are converted into tokens for operators and punctuators:

preprocessing-op-or-punc: one of

```
{     }     [     ]     #     ##    (     )
<:    :>    <%    %>    %:    %:%:  ;     :     ...
new   delete ?     ::    .     .*
+     -     *     /     %     ^     &     |     ~
!     =     <     >     +=    -=    *=    /=    %=
^=    &=    |=    <<    >>    >>=   <<=   ==    !=
<=    >=    &&    ||    ++    --    ,     ->*   ->
and   and_eq bitand bitor compl  not   not_eq
or    or_eq xor    xor_eq
```

Each *preprocessing-op-or-punc* is converted to a single token in translation phase 7 (2.6).

2.299 Literals [lex.literal]

2.299.1 Kinds of literals [lex.literal.kinds]

1

There are several kinds of literals.[21]

21) The term "literal" generally designates, in this International Standard, those tokens that are called "constants" in ISO C.

literal:
- *integer-literal*
- *character-literal*
- *floating-literal*
- *string-literal*
- *boolean-literal*
- *pointer-literal*
- *user-defined-literal*

2.300.1 Integer literals [lex.icon]

integer-literal:
- *binary-literal integer-suffix*$_{opt}$
- *octal-literal integer-suffix*$_{opt}$
- *decimal-literal integer-suffix*$_{opt}$
- *hexadecimal-literal integer-suffix*$_{opt}$

binary-literal:
- `0b` *binary-digit*
- `0B` *binary-digit*
- *binary-literal* `'`$_{opt}$ *binary-digit*

octal-literal:
- `0`
- *octal-literal* `'`$_{opt}$ *octal-digit*

decimal-literal:
- *nonzero-digit*
- *decimal-literal* `'`$_{opt}$ *digit*

hexadecimal-literal:
- `0x` *hexadecimal-digit*
- `0X` *hexadecimal-digit*
- *hexadecimal-literal* `'`$_{opt}$ *hexadecimal-digit*

binary-digit:
- `0`
- `1`

octal-digit: one of
- `0 1 2 3 4 5 6 7`

nonzero-digit: one of
- `1 2 3 4 5 6 7 8 9`

hexadecimal-digit: one of
- `0 1 2 3 4 5 6 7 8 9`
- `a b c d e f`
- `A B C D E F`

integer-suffix:
- *unsigned-suffix long-suffix*$_{opt}$
- *unsigned-suffix long-long-suffix*$_{opt}$
- *long-suffix unsigned-suffix*$_{opt}$
- *long-long-suffix unsigned-suffix*$_{opt}$

unsigned-suffix: one of
- `u U`

long-suffix: one of
- `l L`

long-long-suffix: one of
- `ll LL`

1 An *integer literal* is a sequence of digits that has no period or exponent part, with optional separating single

quotes that are ignored when determining its value. An integer literal may have a prefix that specifies its base and a suffix that specifies its type. The lexically first digit of the sequence of digits is the most significant. A *binary* integer literal (base two) begins with `0b` or `0B` and consists of a sequence of binary digits. An *octal* integer literal (base eight) begins with the digit `0` and consists of a sequence of octal digits.[22] A *decimal* integer literal (base ten) begins with a digit other than `0` and consists of a sequence of decimal digits. A *hexadecimal* integer literal (base sixteen) begins with `0x` or `0X` and consists of a sequence of hexadecimal digits, which include the decimal digits and the letters `a` through `f` and `A` through `F` with decimal values ten through fifteen. [*Example:* The number twelve can be written `12`, `014`, `0XC`, or `0b1100`. The literals `1048576`, `1'048'576`, `0X100000`, `0x10'0000`, and `0'004'000'000` all have the same value. — *end example*]

1

The type of an integer literal is the first of the corresponding list in Table 6 in which its value can be represented.

Table 6 — Types of integer literals

Suffix	Decimal literal	Binary, octal, or hexadecimal literal

none	`int`
	`long int`
	`long long int`
		`int`
		`unsigned int`
		`long int`
		`unsigned long int`
		`long long int`
		`unsigned long long int`

`u` or `U`	`unsigned int`
	`unsigned long int`
	`unsigned long long int`
		`unsigned int`
		`unsigned long int`
		`unsigned long long int`

`l` or `L`	`long int`
	`long long int`
		`long int`
		`unsigned long int`
		`long long int`
		`unsigned long long int`

Both `u` or `U` and `l` or `L`	`unsigned long int` `unsigned long long int`	`unsigned long int` `unsigned long long int`
`ll` or `LL`	`long long int`	`long long int` `unsigned long long int`
Both `u` or `U` and `ll` or `LL`	`unsigned long long int`	`unsigned long long int`

1

If an integer literal cannot be represented by any type in its list and an extended integer type (3.3151.1) can represent its value, it may have that extended integer type. If all of the types in the list for the literal are signed, the extended integer type shall be signed. If all of the types in the list for the literal are unsigned, the extended integer type shall be unsigned. If the list contains both signed and unsigned types, the extended integer type may be signed or unsigned. A program is ill-formed if one of its translation units contains an integer literal that cannot be represented by any of the allowed types.

22) The digits `8` and `9` are not octal digits.

2.356.1 Character literals [lex.ccon]

character-literal:
  ' *c-char-sequence* '
  u' *c-char-sequence* '
  U' *c-char-sequence* '
  L' *c-char-sequence* '

c-char-sequence:
  c-char
  c-char-sequence c-char

c-char:
  any member of the source character set except
    the single-quote ', backslash \, or new-line character
  escape-sequence
  universal-character-name

escape-sequence:
  simple-escape-sequence
  octal-escape-sequence
  hexadecimal-escape-sequence

simple-escape-sequence: one of
  `\'  \"  \?  \\`
  `\a  \b  \f  \n  \r  \t  \v`

octal-escape-sequence:
  \ *octal-digit*
  \ *octal-digit octal-digit*
  \ *octal-digit octal-digit octal-digit*

hexadecimal-escape-sequence:
  `\x` *hexadecimal-digit*
  hexadecimal-escape-sequence hexadecimal-digit

1

A character literal is one or more characters enclosed in single quotes, as in `'x'`, optionally preceded by one of the letters `u`, `U`, or `L`, as in `u'y'`, `U'z'`, or `L'x'`, respectively. A character literal that does not begin with `u`, `U`, or `L` is an ordinary character literal, also referred to as a narrow-character literal. An ordinary character literal that contains a single *c-char* representable in the execution character set has type `char`, with value equal to the numerical value of the encoding of the *c-char* in the execution character set. An ordinary character literal that contains more than one *c-char* is a *multicharacter literal*. A multicharacter literal, or an ordinary character literal containing a single *c-char* not representable in the execution character set, is conditionally-supported, has type `int`, and has an implementation-defined value.

1

A character literal that begins with the letter `u`, such as `u'y'`, is a character literal of type `char16_t`. The value of a `char16_t` literal containing a single *c-char* is equal to its ISO 10646 code point value, provided that the code point is representable with a single 16-bit code unit. (That is, provided it is a basic multi-lingual plane code point.) If the value is not representable within 16 bits, the program is ill-formed. A `char16_t` literal containing multiple *c-char*s is ill-formed. A character literal that begins with the letter `U`, such as `U'z'`, is a character literal of type `char32_t`. The value of a `char32_t` literal containing a single *c-char* is equal to its ISO 10646 code point value. A `char32_t` literal containing multiple *c-char*s is ill-formed. A character literal that begins with the letter `L`, such as `L'x'`, is a wide-character literal. A wide-character literal has type `wchar_t`.[23] The value of a wide-character literal containing a single *c-char* has value equal to the numerical value of the encoding of the *c-char* in the execution wide-character set, unless the *c-char* has no representation in the execution wide-character set, in which case the value is implementation-defined.

23) They are intended for character sets where a character does not fit into a single byte.

[*Note:* The type `wchar_t` is able to represent all members of the execution wide-character set (see 3.3151.1). — *end note*]. The value of a wide-character literal containing multiple *c-char*s is implementation-defined.

1

Certain nongraphic characters, the single quote `'`, the double quote `"`, the question mark `?`,[24] and the backslash `\`, can be represented according to Table 7. The double quote `"` and the question mark `?`, can be represented as themselves or by the escape sequences `\"` and `\?` respectively, but the single quote `'` and the backslash `\` shall be represented by the escape sequences `\'` and `\\` respectively. Escape sequences in which the character following the backslash is not listed in Table 7 are conditionally-supported, with implementation-defined semantics. An escape sequence specifies a single character.

Table 7 — Escape sequences

new-line	NL(LF)	`\n`
horizontal tab	HT	`\t`
vertical tab	VT	`\v`
backspace	BS	`\b`
carriage return	CR	`\r`
form feed	FF	`\f`
alert	BEL	`\a`
backslash	\	`\\`
question mark	?	`\?`
single quote	'	`\'`
double quote	"	`\"`
octal number	*ooo*	`\`*ooo*
hex number	*hhh*	`\x`*hhh*

1

The escape `\`*ooo* consists of the backslash followed by one, two, or three octal digits that are taken to specify the value of the desired character. The escape `\x`*hhh* consists of the backslash followed by `x` followed by one or more hexadecimal digits that are taken to specify the value of the desired character. There is no limit to the number of digits in a hexadecimal sequence. A sequence of octal or hexadecimal digits is terminated by the first character that is not an octal digit or a hexadecimal digit, respectively. The value of a character literal is implementation-defined if it falls outside of the implementation-defined range defined for `char` (for literals with no prefix), `char16_t` (for literals prefixed by `'u'`), `char32_t` (for literals prefixed by `'U'`), or `wchar_t` (for literals prefixed by `'L'`).

1

A universal-character-name is translated to the encoding, in the appropriate execution character set, of the character named. If there is no such encoding, the universal-character-name is translated to an implementation-defined encoding. [*Note:* In translation phase 1, a universal-character-name is introduced whenever an actual extended character is encountered in the source text. Therefore, all extended characters are described in terms of universal-character-names. However, the actual compiler implementation may use its own native character set, so long as the same results are obtained. — *end note*]

2.431.1 Floating literals [lex.fcon]

floating-literal:
  fractional-constant exponent-part$_{opt}$ *floating-suffix*$_{opt}$
  digit-sequence exponent-part floating-suffix$_{opt}$

24) Using an escape sequence for a question mark can avoid accidentally creating a trigraph.

fractional-constant:
  digit-sequence$_{opt}$ `.` *digit-sequence*
  digit-sequence `.`
exponent-part:
  `e` *sign*$_{opt}$ *digit-sequence*
  `E` *sign*$_{opt}$ *digit-sequence*
sign: one of
  `+ -`
digit-sequence:
  digit
  digit-sequence `'`$_{opt}$ *digit*
floating-suffix: one of
  `f l F L`

1

A floating literal consists of an integer part, a decimal point, a fraction part, an `e` or `E`, an optionally signed integer exponent, and an optional type suffix. The integer and fraction parts both consist of a sequence of decimal (base ten) digits. Optional separating single quotes in a *digit-sequence* are ignored when determining its value. [*Example:* The literals `1.602'176'565e-19` and `1.602176565e-19` have the same value. — *end example*] Either the integer part or the fraction part (not both) can be omitted; either the decimal point or the letter `e` (or `E`) and the exponent (not both) can be omitted. The integer part, the optional decimal point and the optional fraction part form the *significant part* of the floating literal. The exponent, if present, indicates the power of 10 by which the significant part is to be scaled. If the scaled value is in the range of representable values for its type, the result is the scaled value if representable, else the larger or smaller representable value nearest the scaled value, chosen in an implementation-defined manner. The type of a floating literal is `double` unless explicitly specified by a suffix. The suffixes `f` and `F` specify `float`, the suffixes `l` and `L` specify `long double`. If the scaled value is not in the range of representable values for its type, the program is ill-formed.

2.453.1 String literals [lex.string]

string-literal:
  encoding-prefix$_{opt}$`"` *s-char-sequence*$_{opt}$`"`
  encoding-prefix$_{opt}$`R` *raw-string*
encoding-prefix:
  `u8`
  `u`
  `U`
  `L`
s-char-sequence:
  s-char
  s-char-sequence s-char
s-char:
  any member of the source character set except
    the double-quote `"`, backslash `\`, or new-line character
  escape-sequence
  universal-character-name
raw-string:
  `"` *d-char-sequence*$_{opt}$`(` *r-char-sequence*$_{opt}$`)` *d-char-sequence*$_{opt}$`"`
r-char-sequence:
  r-char
  r-char-sequence r-char

r-char:
any member of the source character set, except
a right parenthesis `)` followed by the initial *d-char-sequence*
(which may be empty) followed by a double quote `"`.

d-char-sequence:
d-char
d-char-sequence d-char

d-char:
any member of the basic source character set except:
space, the left parenthesis `(`, the right parenthesis `)`, the backslash `\`,
and the control characters representing horizontal tab,
vertical tab, form feed, and newline.

1

A string literal is a sequence of characters (as defined in 2.356.1) surrounded by double quotes, optionally prefixed by `R`, `u8`, `u8R`, `u`, `uR`, `U`, `UR`, `L`, or `LR`, as in `"..."`, `R"(...)"`, `u8"..."`, `u8R"**(...)**"`, `u"..."`, `uR"*~(...)*~"`, `U"..."`, `UR"zzz(...)zzz"`, `L"..."`, or `LR"(...)"`, respectively.

1

A string literal that has an `R` in the prefix is a *raw string literal*. The *d-char-sequence* serves as a delimiter. The terminating *d-char-sequence* of a *raw-string* is the same sequence of characters as the initial *d-char-sequence*. A *d-char-sequence* shall consist of at most 16 characters.

1 [*Note:* The characters `'('` and `')'` are permitted in a *raw-string*. Thus, `R"delimiter((a|b))delimiter"` is equivalent to `"(a|b)"`. — *end note*]

2

[*Note:* A source-file new-line in a raw string literal results in a new-line in the resulting execution *string-literal*. Assuming no whitespace at the beginning of lines in the following example, the assert will succeed:

```
const char* p = R"(a\
b
c)";
assert(std::strcmp(p, "a\\\nb\nc") == 0);
```

— *end note*]

1 [*Example:* The raw string

```
R"a(
)\
a"
)a"
```

is equivalent to `"\n)\\\na\"\n"`. The raw string

```
R"(??)"
```

is equivalent to `"\?\?"`. The raw string

```
R"#(
)??="
)#"
```

is equivalent to `"\n)\?\?=\"\n"`. — *end example*]

2

After translation phase 6, a string literal that does not begin with an *encoding-prefix* is an ordinary string literal, and is initialized with the given characters.

1 A string literal that begins with `u8`, such as `u8"asdf"`, is a UTF-8 string literal.

2

Ordinary string literals and UTF-8 string literals are also referred to as narrow string literals. A narrow string literal has type "array of n `const char`", where n is the size of the string as defined below, and has static storage duration (3.2235).

1 For a UTF-8 string literal, each successive element of the object representation (3.2955) has the value of the corresponding code unit of the UTF-8 encoding of the string.

2

A string literal that begins with `u`, such as `u"asdf"`, is a `char16_t` string literal. A `char16_t` string literal has type "array of n `const char16_t`", where n is the size of the string as defined below; it has static storage duration and is initialized with the given characters. A single *c-char* may produce more than one `char16_t` character in the form of surrogate pairs.

1 A string literal that begins with `U`, such as `U"asdf"`, is a `char32_t` string literal. A `char32_t` string literal has type "array of n `const char32_t`", where n is the size of the string as defined below; it has static storage duration and is initialized with the given characters.

2

A string literal that begins with `L`, such as `L"asdf"`, is a wide string literal. A wide string literal has type "array of n `const wchar_t`", where n is the size of the string as defined below; it has static storage duration and is initialized with the given characters.

1

Whether all string literals are distinct (that is, are stored in nonoverlapping objects) is implementation-defined. The effect of attempting to modify a string literal is undefined.

1

In translation phase 6 (2.6), adjacent string literals are concatenated. If both string literals have the same *encoding-prefix*, the resulting concatenated string literal has that *encoding-prefix*. If one string literal has no *encoding-prefix*, it is treated as a string literal of the same *encoding-prefix* as the other operand. If a UTF-8 string literal token is adjacent to a wide string literal token, the program is ill-formed. Any other concatenations are conditionally-supported with implementation-defined behavior. [*Note:* This concatenation is an interpretation, not a conversion. Because the interpretation happens in translation phase 6 (after each character from a literal has been translated into a value from the appropriate character set), a string literal's initial rawness has no effect on the interpretation or well-formedness of the concatenation. — *end note*] Table 8 has some examples of valid concatenations.

Table 8 — String literal concatenations

Source		Means	Source		Means	Source		Means
`u"a"`	`u"b"`	`u"ab"`	`U"a"`	`U"b"`	`U"ab"`	`L"a"`	`L"b"`	`L"ab"`
`u"a"`	`"b"`	`u"ab"`	`U"a"`	`"b"`	`U"ab"`	`L"a"`	`"b"`	`L"ab"`
`"a"`	`u"b"`	`u"ab"`	`"a"`	`U"b"`	`U"ab"`	`"a"`	`L"b"`	`L"ab"`

Characters in concatenated strings are kept distinct.

[*Example:*

```
"\xA" "B"
```

contains the two characters `'\xA'` and `'B'` after concatenation (and not the single hexadecimal character `'\xAB'`). — *end example*]

1

After any necessary concatenation, in translation phase 7 (2.6), `'\0'` is appended to every string literal so that programs that scan a string can find its end.

1

Escape sequences and universal-character-names in non-raw string literals have the same meaning as in character literals (2.356.1), except that the single quote `'` is representable either by itself or by the escape sequence `\'`, and the double quote `"` shall be preceded by a `\`. In a narrow string literal, a universal-character-name may map to more than one `char` element due to *multibyte encoding*. The size of a `char32_t` or wide string literal is the total number of escape sequences, universal-character-names, and other characters, plus one for the terminating `U'\0'` or `L'\0'`. The size of a `char16_t` string literal is the total number of escape sequences, universal-character-names, and other characters, plus one for each character requiring a surrogate pair, plus one for the terminating `u'\0'`. [*Note:* The size of a `char16_t` string literal is the number of code units, not the number of characters. — *end note*] Within `char32_t` and `char16_t` literals, any universal-character-names shall be within the range `0x0` to `0x10FFFF`. The size of a narrow string literal is the total number of escape sequences and other characters, plus at least one for the multibyte encoding of each universal-character-name, plus one for the terminating `'\0'`.

2.605.1 Boolean literals [lex.bool]

boolean-literal:
`false`
`true`

1

The Boolean literals are the keywords `false` and `true`. Such literals are prvalues and have type `bool`.

2.607.1 Pointer literals [lex.nullptr]

pointer-literal:
`nullptr`

1

The pointer literal is the keyword `nullptr`. It is a prvalue of type `std::nullptr_t`. [*Note:* `std::nullptr_t` is a distinct type that is neither a pointer type nor a pointer to member type; rather, a prvalue of this type is a null pointer constant and can be converted to a null pointer value or null member pointer value. See 4.579 and 4.710. — *end note*]

2.622.1 User-defined literals [lex.ext]

user-defined-literal:
user-defined-integer-literal
user-defined-floating-literal
user-defined-string-literal
user-defined-character-literal

user-defined-integer-literal:
decimal-literal ud-suffix
octal-literal ud-suffix
hexadecimal-literal ud-suffix
binary-literal ud-suffix

user-defined-floating-literal:
fractional-constant exponent-part$_{opt}$ *ud-suffix*
digit-sequence exponent-part ud-suffix

user-defined-string-literal:
string-literal ud-suffix
user-defined-character-literal:
character-literal ud-suffix
ud-suffix:
identifier

1

If a token matches both *user-defined-literal* and another literal kind, it is treated as the latter. [*Example:* `123_km` is a *user-defined-literal*, but `12LL` is an *integer-literal*. — *end example*] The syntactic non-terminal preceding the *ud-suffix* in a *user-defined-literal* is taken to be the longest sequence of characters that could match that non-terminal.

1

A *user-defined-literal* is treated as a call to a literal operator or literal operator template (13.1241.1). To determine the form of this call for a given *user-defined-literal* L with *ud-suffix* X, the *literal-operator-id* whose literal suffix identifier is X is looked up in the context of L using the rules for unqualified name lookup (3.952.1). Let S be the set of declarations found by this lookup. S shall not be empty.

1

If L is a *user-defined-integer-literal*, let n be the literal without its *ud-suffix*. If S contains a literal operator with parameter type `unsigned long long`, the literal L is treated as a call of the form

```
operator "" X(nULL)
```

Otherwise, S shall contain a raw literal operator or a literal operator template (13.1241.1) but not both. If S contains a raw literal operator, the literal L is treated as a call of the form

```
operator "" X("n")
```

Otherwise (S contains a literal operator template), L is treated as a call of the form

```
operator "" X<'c1', 'c2', ... 'ck'>()
```

where n is the source character sequence $c_1c_2...c_k$. [*Note:* The sequence $c_1c_2...c_k$ can only contain characters from the basic source character set. — *end note*]

1 If L is a *user-defined-floating-literal*, let f be the literal without its *ud-suffix*. If S contains a literal operator with parameter type `long double`, the literal L is treated as a call of the form

```
operator "" X(fL)
```

Otherwise, S shall contain a raw literal operator or a literal operator template (13.1241.1) but not both. If S contains a raw literal operator, the *literal* L is treated as a call of the form

```
operator "" X("f")
```

Otherwise (S contains a literal operator template), L is treated as a call of the form

```
operator "" X<'c1', 'c2', ... 'ck'>()
```

where f is the source character sequence $c_1c_2...c_k$. [*Note:* The sequence $c_1c_2...c_k$ can only contain characters from the basic source character set. — *end note*]

2

If L is a *user-defined-string-literal*, let *str* be the literal without its *ud-suffix* and let *len* be the number of code units in *str* (i.e., its length excluding the terminating null character). The literal L is treated as a call of the form

```
operator "" X(str, len)
```

1 If L is a *user-defined-character-literal*, let *ch* be the literal without its *ud-suffix*. S shall contain a literal operator (13.1241.1) whose only parameter has the type of *ch* and the literal L is treated as a call of the form

```
operator "" X(ch)
```

2 [*Example:*

```
long double operator "" _w(long double);
std::string operator "" _w(const char16_t*, std::size_t);
unsigned operator "" _w(const char*);
int main() {
  1.2_w;      // calls operator "" _w(1.2L)
  u"one"_w;   // calls operator "" _w(u"one", 3)
  12_w;       // calls operator "" _w("12")
  "two"_w;    // error: no applicable literal operator
}
```

— *end example*]

3 In translation phase 6 (2.6), adjacent string literals are concatenated and *user-defined-string-literal*s are considered string literals for that purpose. During concatenation, *ud-suffix*es are removed and ignored and the concatenation process occurs as described in 2.453.1. At the end of phase 6, if a string literal is the result of a concatenation involving at least one *user-defined-string-literal*, all the participating *user-defined-string-literal*s shall have the same *ud-suffix* and that suffix is applied to the result of the concatenation.

4 [*Example:*

```
int main() {
  L"A" "B" "C"_x; // OK: same as L"ABC"_x
  "P"_x "Q" "R"_y;// error: two different ud-suffixes
}
```

— *end example*]

5 Some *identifier*s appearing as *ud-suffix*es are reserved for future standardization (17.424.0.0.1). A program containing such a *ud-suffix* is ill-formed, no diagnostic required.

3 Basic concepts [basic]

1 [*Note:* This Clause presents the basic concepts of the C++ language. It explains the difference between an *object* and a *name* and how they relate to the value categories for expressions. It introduces the concepts of a *declaration* and a *definition* and presents C++'s notion of *type*, *scope*, *linkage*, and *storage duration*. The mechanisms for starting and terminating a program are discussed. Finally, this Clause presents the *fundamental* types of the language and lists the ways of constructing *compound* types from these. — *end note*]

2 [*Note:* This Clause does not cover concepts that affect only a single part of the language. Such concepts are discussed in the relevant Clauses. — *end note*]

3

An *entity* is a value, object, reference, function, enumerator, type, class member, template, template specialization, namespace, parameter pack, or `this`.

1

A *name* is a use of an *identifier* (2.268), *operator-function-id* (13.1122), *literal-operator-id* (13.1241.1), *conversion-function-id* (12.742.1), or *template-id* (14.224) that denotes an entity or *label* (6.413.1, 6.3).

1

Every name that denotes an entity is introduced by a *declaration*. Every name that denotes a label is introduced either by a `goto` statement (6.413.1) or a *labeled-statement* (6.3).

1

A *variable* is introduced by the declaration of a reference other than a non-static data member or of an object. The variable's name, if any, denotes the reference or object.

1

Some names denote types or templates. In general, whenever a name is encountered it is necessary to determine whether that name denotes one of these entities before continuing to parse the program that contains it. The process that determines this is called *name lookup* (3.923).

1

Two names are *the same* if

— they are *identifier*s composed of the same character sequence, or

— they are *operator-function-id*s formed with the same operator, or

— they are *conversion-function-id*s formed with the same type, or

— they are *template-id*s that refer to the same class, function, or variable (14.595), or

— they are the names of literal operators (13.1241.1) formed with the same literal suffix identifier.

1

A name used in more than one translation unit can potentially refer to the same entity in these translation units depending on the linkage (3.1459) of the name specified in each translation unit.

3.43 Declarations and definitions [basic.def]

1

A declaration (Clause 7) may introduce one or more names into a translation unit or redeclare names introduced by previous declarations. If so, the declaration specifies the interpretation and attributes of these names. A declaration may also have effects including:

— a static assertion (Clause 7),

— controlling template instantiation (14.1841.1),

— use of attributes (Clause 7), and

— nothing (in the case of an *empty-declaration*).

1

A declaration is a *definition* unless it declares a function without specifying the function's body (8.706), it contains the `extern` specifier (7.49.1) or a *linkage-specification*[25] (7.1477) and neither an *initializer* nor a *function-body*, it declares a static data member in a class definition (9.194, 9.559), it is a class name declaration (9.172), it is an *opaque-enum-declaration* (7.1001), it is a *template-parameter* (14.97), it is a *parameter-declaration* (8.360.1) in a function declarator that is not the *declarator* of a *function-definition*, or it is a `typedef` declaration (7.235.1), an *alias-declaration* (7.235.1), a *using-declaration* (7.1311.1), a *static_assert-declaration* (Clause 7), an *attribute-declaration* (Clause 7), an *empty-declaration* (Clause 7), or a *using-directive* (7.1416.1).

[*Example:* all but one of the following are definitions:

```
int a;                          // defines a
extern const int c = 1;         // defines c
int f(int x) { return x+a; }    // defines f and defines x
struct S { int a; int b; };     // defines S, S::a, and S::b
struct X {                      // defines X
  int x;                        // defines non-static data member x
  static int y;                 // declares static data member y
  X(): x(0) { }                 // defines a constructor of X
};
int X::y = 1;                   // defines X::y
enum { up, down };              // defines up and down
namespace N { int d; }          // defines N and N::d
namespace N1 = N;               // defines N1
X anX;                          // defines anX
```

whereas these are just declarations:

```
extern int a;                   // declares a
extern const int c;             // declares c
int f(int);                     // declares f
struct S;                       // declares S
typedef int Int;                // declares Int
extern X anotherX;              // declares anotherX
using N::d;                     // declares d
```

— *end example*]

1

25) Appearing inside the braced-enclosed *declaration-seq* in a *linkage-specification* does not affect whether a declaration is a definition.

[*Note:* In some circumstances, C++ implementations implicitly define the default constructor (12.30), copy constructor (12.1577), move constructor (12.1577), copy assignment operator (12.1577), move assignment operator (12.1577), or destructor (12.807) member functions. — *end note*] [*Example:* given

```
#include <string>

struct C {
  std::string s;              // std::string is the standard library class (Clause 21)
};

int main() {
  C a;
  C b = a;
  b = a;
}
```

the implementation will implicitly define functions to make the definition of `C` equivalent to

```
struct C {
  std::string s;
  C() : s() { }
  C(const C& x): s(x.s) { }
  C(C&& x): s(static_cast<std::string&&>(x.s)) { }
    // :  s(std::move(x.s)) { }
  C& operator=(const C& x) { s = x.s; return *this; }
  C& operator=(C&& x) { s = static_cast<std::string&&>(x.s); return *this; }
    // { s = std::move(x.s); return *this; }
  ~C() { }
};
```

— *end example*]

1

[*Note:* A class name can also be implicitly declared by an *elaborated-type-specifier* (7.808.0.1). — *end note*]

1

A program is ill-formed if the definition of any object gives the object an incomplete type (3.2955).

3.91 One definition rule [basic.def.odr]

1

No translation unit shall contain more than one definition of any variable, function, class type, enumeration type, or template.

1

An expression is *potentially evaluated* unless it is an unevaluated operand (Clause 5) or a subexpression thereof. The set of *potential results* of an expression `e` is defined as follows:

— If `e` is an *id-expression* (5.398.1), the set contains only `e`.

— If `e` is a class member access expression (5.1194.1), the set contains the potential results of the object expression.

— If `e` is a pointer-to-member expression (5.2832) whose second operand is a constant expression, the set contains the potential results of the object expression.

— If `e` has the form `(e1)`, the set contains the potential results of `e1`.

— If `e` is a glvalue conditional expression (5.3337), the set is the union of the sets of potential results of the second and third operands.

— If `e` is a comma expression (5.3602), the set contains the potential results of the right operand.

— Otherwise, the set is empty.

[*Note:* This set is a (possibly-empty) set of *id-expressions*, each of which is either `e` or a subexpression of `e`. [*Example:* In the following example, the set of potential results of the initializer of `n` contains the first `S::x` subexpression, but not the second `S::x` subexpression.

```
struct S { static const int x = 0; };
const int &f(const int &r);
int n = b ? (1, S::x)  // S::x is not odr-used here
          : f(S::x);   // S::x is odr-used here, so
                       // a definition is required
```

— *end example*] — *end note*]

1

A variable `x` whose name appears as a potentially-evaluated expression `ex` is *odr-used* by `ex` unless applying the lvalue-to-rvalue conversion (4.127) to `x` yields a constant expression (5.3643) that does not invoke any non-trivial functions and, if `x` is an object, `ex` is an element of the set of potential results of an expression `e`, where either the lvalue-to-rvalue conversion (4.127) is applied to `e`, or `e` is a discarded-value expression (Clause 5). `this` is odr-used if it appears as a potentially-evaluated expression (including as the result of the implicit transformation in the body of a non-static member function (9.465.1)). A virtual member function is odr-used if it is not pure. A function whose name appears as a potentially-evaluated expression is odr-used if it is the unique lookup result or the selected member of a set of overloaded functions (3.923, 13.88, 13.1084), unless it is a pure virtual function and its name is not explicitly qualified. [*Note:* This covers calls to named functions (5.955.1), operator overloading (Clause 13), user-defined conversions (12.742.1), allocation function for placement new (5.2221.1), as well as non-default initialization (8.821). A constructor selected to copy or move an object of class type is odr-used even if the call is actually elided by the implementation (12.1577). — *end note*] An allocation or deallocation function for a class is odr-used by a new expression appearing in a potentially-evaluated expression as specified in 5.2221.1 and 12.988. A deallocation function for a class is odr-used by a delete expression appearing in a potentially-evaluated expression as specified in 5.2455.1 and 12.988. A non-placement allocation or deallocation function for a class is odr-used by the definition of a constructor of that class. A non-placement deallocation function for a class is odr-used by the definition of the destructor of that class, or by being selected by the lookup at the point of definition of a virtual destructor (12.807).[26] An assignment operator function in a class is odr-used by an implicitly-defined copy-assignment or move-assignment function for another class as specified in 12.1577. A default constructor for a class is odr-used by default initialization or value initialization as specified in 8.821. A constructor for a class is odr-used as specified in 8.821. A destructor for a class is odr-used if it is potentially invoked (12.807).

1

Every program shall contain exactly one definition of every non-inline function or variable that is odr-used in that program; no diagnostic required. The definition can appear explicitly in the program, it can be found in the standard or a user-defined library, or (when appropriate) it is implicitly defined (see 12.30, 12.807 and 12.1577). An inline function shall be defined in every translation unit in which it is odr-used.

1

26) An implementation is not required to call allocation and deallocation functions from constructors or destructors; however, this is a permissible implementation technique.

Exactly one definition of a class is required in a translation unit if the class is used in a way that requires the class type to be complete. [*Example:* the following complete translation unit is well-formed, even though it never defines `X`:

```
struct X;                    // declare X as a struct type
struct X* x1;                // use X in pointer formation
X* x2;                       // use X in pointer formation
```

— *end example*] [*Note:* The rules for declarations and expressions describe in which contexts complete class types are required. A class type `T` must be complete if:

— an object of type `T` is defined (3.43), or

— a non-static class data member of type `T` is declared (9.194), or

— `T` is used as the object type or array element type in a *new-expression* (5.2221.1), or

— an lvalue-to-rvalue conversion is applied to a glvalue referring to an object of type `T` (4.127), or

— an expression is converted (either implicitly or explicitly) to type `T` (Clause 4, 5.1108.1, 5.1295.1, 5.1443.1, 5.2765), or

— an expression that is not a null pointer constant, and has type other than *cv* `void*`, is converted to the type pointer to `T` or reference to `T` using a standard conversion (Clause 4), a `dynamic_cast` (5.1295.1) or a `static_cast` (5.1443.1), or

— a class member access operator is applied to an expression of type `T` (5.1194.1), or

— the `typeid` operator (5.1393.1) or the `sizeof` operator (5.2075.1) is applied to an operand of type `T`, or

— a function with a return type or argument type of type `T` is defined (3.43) or called (5.955.1), or

— a class with a base class of type `T` is defined (Clause 10), or

— an lvalue of type `T` is assigned to (5.3483), or

— the type `T` is the subject of an `alignof` expression (5.2747.1), or

— an *exception-declaration* has type `T`, reference to `T`, or pointer to `T` (15.146).

— *end note*]

1

There can be more than one definition of a class type (Clause 9), enumeration type (7.1001), inline function with external linkage (7.94.1), class template (Clause 14), non-static function template (14.972.1), static data member of a class template (14.634.0.1), member function of a class template (14.630.0.1), or template specialization for which some template parameters are not specified (14.1614, 14.812.1) in a program provided that each definition appears in a different translation unit, and provided the definitions satisfy the following requirements. Given such an entity named `D` defined in more than one translation unit, then

— each definition of `D` shall consist of the same sequence of tokens; and

— in each definition of D, corresponding names, looked up according to 3.923, shall refer to an entity defined within the definition of D, or shall refer to the same entity, after overload resolution (13.88) and after matching of partial template specialization (14.2662.1), except that a name can refer to a non-volatile const object with internal or no linkage if the object has the same literal type in all definitions of D, and the object is initialized with a constant expression (5.3643), and the object is not odr-used, and the object has the same value in all definitions of D; and

— in each definition of D, corresponding entities shall have the same language linkage; and

— in each definition of D, the overloaded operators referred to, the implicit calls to conversion functions, constructors, operator new functions and operator delete functions, shall refer to the same function, or to a function defined within the definition of D; and

— in each definition of D, a default argument used by an (implicit or explicit) function call is treated as if its token sequence were present in the definition of D; that is, the default argument is subject to the three requirements described above (and, if the default argument has sub-expressions with default arguments, this requirement applies recursively).[27]

— if D is a class with an implicitly-declared constructor (12.30), it is as if the constructor was implicitly defined in every translation unit where it is odr-used, and the implicit definition in every translation unit shall call the same constructor for a base class or a class member of D. [*Example:*

```
//translation unit 1:
struct X {
  X(int);
  X(int, int);
};
X::X(int = 0) { }
class D: public X { };
D d2;                          // X(int) called by D()

//translation unit 2:
struct X {
  X(int);
  X(int, int);
};
X::X(int = 0, int = 0) { }
class D: public X { };         // X(int, int) called by D();
                               // D()'s implicit definition
                               // violates the ODR
```

— *end example*]

If D is a template and is defined in more than one translation unit, then the preceding requirements shall apply both to names from the template's enclosing scope used in the template definition (14.1444.1), and also to dependent names at the point of instantiation (14.1257.1). If the definitions of D satisfy all these requirements, then the behavior is as if there were a single definition of D. If the definitions of D do not satisfy these requirements, then the behavior is undefined.

3.464 Scope [basic.scope]

3.464.1 Declarative regions and scopes [basic.scope.declarative]

1

27) 8.545.1 describes how default argument names are looked up.

Every name is introduced in some portion of program text called a *declarative region*, which is the largest part of the program in which that name is *valid*, that is, in which that name may be used as an unqualified name to refer to the same entity. In general, each particular name is valid only within some possibly discontiguous portion of program text called its *scope*. To determine the scope of a declaration, it is sometimes convenient to refer to the *potential scope* of a declaration. The scope of a declaration is the same as its potential scope unless the potential scope contains another declaration of the same name. In that case, the potential scope of the declaration in the inner (contained) declarative region is excluded from the scope of the declaration in the outer (containing) declarative region.

1

[*Example:* in

```
int j = 24;
int main() {
  int i = j, j;
  j = 42;
}
```

the identifier `j` is declared twice as a name (and used twice). The declarative region of the first `j` includes the entire example. The potential scope of the first `j` begins immediately after that `j` and extends to the end of the program, but its (actual) scope excludes the text between the `,` and the `}`. The declarative region of the second declaration of `j` (the `j` immediately before the semicolon) includes all the text between `{` and `}`, but its potential scope excludes the declaration of `i`. The scope of the second declaration of `j` is the same as its potential scope. — *end example*]

1

The names declared by a declaration are introduced into the scope in which the declaration occurs, except that the presence of a `friend` specifier (11.94), certain uses of the *elaborated-type-specifier* (7.808.0.1), and *using-directives* (7.1416.1) alter this general behavior.

1

Given a set of declarations in a single declarative region, each of which specifies the same unqualified name,

— they shall all refer to the same entity, or all refer to functions and function templates; or

— exactly one declaration shall declare a class name or enumeration name that is not a typedef name and the other declarations shall all refer to the same variable or enumerator, or all refer to functions and function templates; in this case the class name or enumeration name is hidden (3.875.1). [*Note:* A namespace name or a class template name must be unique in its declarative region (7.1306.1, Clause 14). — *end note*]

[*Note:* These restrictions apply to the declarative region into which a name is introduced, which is not necessarily the same as the region in which the declaration occurs. In particular, *elaborated-type-specifiers* (7.808.0.1) and friend declarations (11.94) may introduce a (possibly not visible) name into an enclosing namespace; these restrictions apply to that region. Local extern declarations (3.1459) may introduce a name into the declarative region where the declaration appears and also introduce a (possibly not visible) name into an enclosing namespace; these restrictions apply to both regions. — *end note*]

1 [*Note:* The name lookup rules are summarized in 3.923. — *end note*]

3.536.1 Point of declaration [basic.scope.pdecl]

1

The *point of declaration* for a name is immediately after its complete declarator (Clause 8) and before its *initializer* (if any), except as noted below. [*Example:*

```
unsigned char x = 12;
{ unsigned char x = x; }
```

Here the second `x` is initialized with its own (indeterminate) value. — *end example*]

1

[*Note:* a name from an outer scope remains visible up to the point of declaration of the name that hides it.[*Example:*

```
const int  i = 2;
{ int  i[i]; }
```

declares a block-scope array of two integers. — *end example*] — *end note*]

1

The point of declaration for a class or class template first declared by a *class-specifier* is immediately after the *identifier* or *simple-template-id* (if any) in its *class-head* (Clause 9). The point of declaration for an enumeration is immediately after the *identifier* (if any) in either its *enum-specifier* (7.1001) or its first *opaque-enum-declaration* (7.1001), whichever comes first. The point of declaration of an alias or alias template immediately follows the *type-id* to which the alias refers.

1 The point of declaration of a *using-declaration* that does not name a constructor is immediately after the *using-declaration* (7.1311.1).

2

The point of declaration for an enumerator is immediately after its *enumerator-definition*.[*Example:*

```
const int x = 12;
{ enum { x = x }; }
```

Here, the enumerator `x` is initialized with the value of the constant `x`, namely 12. — *end example*]

1

After the point of declaration of a class member, the member name can be looked up in the scope of its class. [*Note:* this is true even if the class is an incomplete class. For example,

```
struct X {
  enum E { z = 16 };
  int b[X::z];      // OK
};
```

— *end note*]

1

The point of declaration of a class first declared in an *elaborated-type-specifier* is as follows:

— for a declaration of the form

class-key attribute-specifier-seq$_{opt}$ *identifier* ;

the *identifier* is declared to be a *class-name* in the scope that contains the declaration, otherwise

— for an *elaborated-type-specifier* of the form

class-key identifier

if the *elaborated-type-specifier* is used in the *decl-specifier-seq* or *parameter-declaration-clause* of a function defined in namespace scope, the *identifier* is declared as a *class-name* in the namespace that contains the declaration; otherwise, except as a friend declaration, the *identifier* is declared in the

smallest namespace or block scope that contains the declaration. [*Note:* These rules also apply within templates. — *end note*] [*Note:* Other forms of *elaborated-type-specifier* do not declare a new name, and therefore must refer to an existing *type-name*. See 3.1395.1 and 7.808.0.1. — *end note*]

1

The point of declaration for an *injected-class-name* (Clause 9) is immediately following the opening brace of the class definition.

1 The point of declaration for a function-local predefined variable (8.706) is immediately before the *function-body* of a function definition.

2 The point of declaration for a template parameter is immediately after its complete *template-parameter*. [*Example:*

```
typedef unsigned char T;
template<class T
  = T      // lookup finds the typedef name of unsigned char
  , T      // lookup finds the template parameter
    N = 0> struct A { };
```

— *end example*]

3

[*Note:* Friend declarations refer to functions or classes that are members of the nearest enclosing namespace, but they do not introduce new names into that namespace (7.1215.0.1). Function declarations at block scope and variable declarations with the `extern` specifier at block scope refer to declarations that are members of an enclosing namespace, but they do not introduce new names into that scope. — *end note*]

1 [*Note:* For point of instantiation of a template, see 14.1486.0.1. — *end note*]

3.694.1 Block scope [basic.scope.block]

1

A name declared in a block (6.15) is local to that block; it has *block scope*. Its potential scope begins at its point of declaration (3.536.1) and ends at the end of its block. A variable declared at block scope is a *local variable*.

1

The potential scope of a function parameter name (including one appearing in a *lambda-declarator*) or of a function-local predefined variable in a function definition (8.706) begins at its point of declaration. If the function has a *function-try-block* the potential scope of a parameter or of a function-local predefined variable ends at the end of the last associated handler, otherwise it ends at the end of the outermost block of the function definition. A parameter name shall not be redeclared in the outermost block of the function definition nor in the outermost block of any handler associated with a *function-try-block*.

1

The name declared in an *exception-declaration* is local to the *handler* and shall not be redeclared in the outermost block of the *handler*.

1

Names declared in the *for-init-statement*, the *for-range-declaration*, and in the *condition* of `if`, `while`, `for`, and `switch` statements are local to the `if`, `while`, `for`, or `switch` statement (including the controlled statement), and shall not be redeclared in a subsequent condition of that statement nor in the outermost block (or, for the `if` statement, any of the outermost blocks) of the controlled statement; see 6.23.

3.729.1 Function prototype scope [basic.scope.proto]

1

In a function declaration, or in any function declarator except the declarator of a function definition (8.706), names of parameters (if supplied) have function prototype scope, which terminates at the end of the nearest enclosing function declarator.

3.735.1 Function scope [basic.funscope]

1

Labels (6.3) have *function scope* and may be used anywhere in the function in which they are declared. Only labels have function scope.

3.736.1 Namespace scope [basic.scope.namespace]

1

The declarative region of a *namespace-definition* is its *namespace-body*. The potential scope denoted by an *original-namespace-name* is the concatenation of the declarative regions established by each of the *namespace-definition*s in the same declarative region with that *original-namespace-name*. Entities declared in a *namespace-body* are said to be *members* of the namespace, and names introduced by these declarations into the declarative region of the namespace are said to be *member names* of the namespace. A namespace member name has namespace scope. Its potential scope includes its namespace from the name's point of declaration (3.536.1) onwards; and for each *using-directive* (7.1416.1) that nominates the member's namespace, the member's potential scope includes that portion of the potential scope of the *using-directive* that follows the member's point of declaration. [*Example:*

```
namespace N {
  int i;
  int g(int a) { return a; }
  int j();
  void q();
}
namespace { int l=1; }
// the potential scope of l is from its point of declaration
// to the end of the translation unit

namespace N {
  int g(char a) {   // overloads N::g(int)
    return l+a;     // l is from unnamed namespace
  }

  int i;            // error: duplicate definition
  int j();          // OK: duplicate function declaration

  int j() {         // OK: definition of N::j()
    return g(i);    // calls N::g(int)
  }
  int q();          // error: different return type
}
```

— *end example*]

1

A namespace member can also be referred to after the `::` scope resolution operator (5.398) applied to the name of its namespace or the name of a namespace which nominates the member's namespace in a

using-directive; see 3.1363.0.1.

1

The outermost declarative region of a translation unit is also a namespace, called the *global namespace*. A name declared in the global namespace has *global namespace scope* (also called *global scope*). The potential scope of such a name begins at its point of declaration (3.536.1) and ends at the end of the translation unit that is its declarative region. A name with global namespace scope is said to be a *global name*.

3.760.1 Class scope [basic.scope.class]

1

The following rules describe the scope of names declared in classes.

1) The potential scope of a name declared in a class consists not only of the declarative region following the name's point of declaration, but also of all function bodies, default arguments, *exception-specification*s, and *brace-or-equal-initializers* of non-static data members in that class (including such things in nested classes).

2) A name `N` used in a class `S` shall refer to the same declaration in its context and when re-evaluated in the completed scope of `S`. No diagnostic is required for a violation of this rule.

3) If reordering member declarations in a class yields an alternate valid program under (1) and (2), the program is ill-formed, no diagnostic is required.

4) A name declared within a member function hides a declaration of the same name whose scope extends to or past the end of the member function's class.

5) The potential scope of a declaration that extends to or past the end of a class definition also extends to the regions defined by its member definitions, even if the members are defined lexically outside the class (this includes static data member definitions, nested class definitions, and member function definitions, including the member function body and any portion of the declarator part of such definitions which follows the *declarator-id*, including a *parameter-declaration-clause* and any default arguments (8.545.1)).[*Example:*

```
typedef int  c;
enum { i = 1 };

class X {
  char  v[i];                       // error: i refers to ::i
                                    // but when reevaluated is X::i
  int  f() { return sizeof(c); }    // OK: X::c
  char  c;
  enum { i = 2 };
};

typedef char*  T;
struct Y {
  T  a;                             // error: T refers to ::T
                                    // but when reevaluated is Y::T
  typedef long  T;
  T  b;
};

typedef int I;
class D {
  typedef I I;                      // error, even though no reordering involved
```

```
};
```

— *end example*]

1

The name of a class member shall only be used as follows:

— in the scope of its class (as described above) or a class derived (Clause 10) from its class,

— after the `.` operator applied to an expression of the type of its class (5.1194.1) or a class derived from its class,

— after the `->` operator applied to a pointer to an object of its class (5.1194.1) or a class derived from its class,

— after the `::` scope resolution operator (5.398) applied to the name of its class or a class derived from its class.

3.865.1 Enumeration scope [basic.scope.enum]

1 The name of a scoped enumerator (7.1001) has *enumeration scope*. Its potential scope begins at its point of declaration and terminates at the end of the *enum-specifier*.

3.865.2 Template parameter scope [basic.scope.temp]

1

The declarative region of the name of a template parameter of a template *template-parameter* is the smallest *template-parameter-list* in which the name was introduced.

1

The declarative region of the name of a template parameter of a template is the smallest *template-declaration* in which the name was introduced. Only template parameter names belong to this declarative region; any other kind of name introduced by the *declaration* of a *template-declaration* is instead introduced into the same declarative region where it would be introduced as a result of a non-template declaration of the same name. [*Example:*

```
namespace N {
  template<class T> struct A { };              // #1
  template<class U> void f(U) { }              // #2
  struct B {
    template<class V> friend int g(struct C*);  // #3
  };
}
```

The declarative regions of `T`, `U` and `V` are the *template-declaration*s on lines `#1`, `#2` and `#3`, respectively. But the names `A`, `f`, `g` and `C` all belong to the same declarative region — namely, the *namespace-body* of `N`. (`g` is still considered to belong to this declarative region in spite of its being hidden during qualified and unqualified name lookup.) — *end example*]

1

The potential scope of a template parameter name begins at its point of declaration (3.536.1) and ends at the end of its declarative region. [*Note:* This implies that a *template-parameter* can be used in the declaration of subsequent *template-parameter*s and their default arguments but cannot be used in preceding *template-parameter*s or their default arguments. For example,

```
template<class T, T* p, class U = T> class X { /* ... */ };
template<class T> void f(T* p = new T);
```

This also implies that a *template-parameter* can be used in the specification of base classes. For example,

```
template<class T> class X : public Array<T> { /* ... */ };
template<class T> class Y : public T { /* ... */ };
```

The use of a template parameter as a base class implies that a class used as a template argument must be defined and not just declared when the class template is instantiated. — *end note*]

1

The declarative region of the name of a template parameter is nested within the immediately-enclosing declarative region. [*Note:* As a result, a *template-parameter* hides any entity with the same name in an enclosing scope (3.875.1). [*Example:*

```
typedef int N;
template<N X, typename N, template<N Y> class T> struct A;
```

Here, `X` is a non-type template parameter of type `int` and `Y` is a non-type template parameter of the same type as the second template parameter of `A`. — *end example*] — *end note*]

1

[*Note:* Because the name of a template parameter cannot be redeclared within its potential scope (14.1177.1), a template parameter's scope is often its potential scope. However, it is still possible for a template parameter name to be hidden; see 14.1177.1. — *end note*]

3.875.1 Name hiding [basic.scope.hiding]

1

A name can be hidden by an explicit declaration of that same name in a nested declarative region or derived class (10.52).

1

A class name (9.172) or enumeration name (7.1001) can be hidden by the name of a variable, data member, function, or enumerator declared in the same scope. If a class or enumeration name and a variable, data member, function, or enumerator are declared in the same scope (in any order) with the same name, the class or enumeration name is hidden wherever the variable, data member, function, or enumerator name is visible.

1

In a member function definition, the declaration of a name at block scope hides the declaration of a member of the class with the same name; see 3.760.1. The declaration of a member in a derived class (Clause 10) hides the declaration of a member of a base class of the same name; see 10.52.

1

During the lookup of a name qualified by a namespace name, declarations that would otherwise be made visible by a *using-directive* can be hidden by declarations with the same name in the namespace containing the *using-directive;* see (3.1363.0.1).

1 If a name is in scope and is not hidden it is said to be *visible*.

3.923 Name lookup [basic.lookup]

1

The name lookup rules apply uniformly to all names (including *typedef-name*s (7.235.1), *namespace-name*s (7.1142), and *class-name*s (9.172)) wherever the grammar allows such names in the context discussed by a particular rule. Name lookup associates the use of a name with a declaration (3.43) of that name. Name lookup shall find an unambiguous declaration for the name (see 10.52). Name lookup may associate more than one

declaration with a name if it finds the name to be a function name; the declarations are said to form a set of overloaded functions (13.12). Overload resolution (13.88) takes place after name lookup has succeeded. The access rules (Clause 11) are considered only once name lookup and function overload resolution (if applicable) have succeeded. Only after name lookup, function overload resolution (if applicable) and access checking have succeeded are the attributes introduced by the name's declaration used further in expression processing (Clause 5).

1

A name "looked up in the context of an expression" is looked up as an unqualified name in the scope where the expression is found.

1

The injected-class-name of a class (Clause 9) is also considered to be a member of that class for the purposes of name hiding and lookup.

1 [*Note:* 3.1459 discusses linkage issues. The notions of scope, point of declaration and name hiding are discussed in 3.464. — *end note*]

3.952.1 Unqualified name lookup [basic.lookup.unqual]

1

In all the cases listed in 3.952.1, the scopes are searched for a declaration in the order listed in each of the respective categories; name lookup ends as soon as a declaration is found for the name. If no declaration is found, the program is ill-formed.

1

The declarations from the namespace nominated by a *using-directive* become visible in a namespace enclosing the *using-directive*; see 7.1416.1. For the purpose of the unqualified name lookup rules described in 3.952.1, the declarations from the namespace nominated by the *using-directive* are considered members of that enclosing namespace.

1

The lookup for an unqualified name used as the *postfix-expression* of a function call is described in 3.1092.1. [*Note:* For purposes of determining (during parsing) whether an expression is a *postfix-expression* for a function call, the usual name lookup rules apply. The rules in 3.1092.1 have no effect on the syntactic interpretation of an expression. For example,

```
typedef int f;
namespace N {
  struct A {
    friend void f(A &);
    operator int();
    void g(A a) {
      int i = f(a);          // f is the typedef, not the friend
                             // function: equivalent to int(a)
    }
  };
}
```

Because the expression is not a function call, the argument-dependent name lookup (3.1092.1) does not apply and the friend function `f` is not found. — *end note*]

1

A name used in global scope, outside of any function, class or user-declared namespace, shall be declared before its use in global scope.

1

A name used in a user-declared namespace outside of the definition of any function or class shall be declared before its use in that namespace or before its use in a namespace enclosing its namespace.

1

A name used in the definition of a function following the function's *declarator-id*[28] that is a member of namespace `N` (where, only for the purpose of exposition, `N` could represent the global scope) shall be declared before its use in the block in which it is used or in one of its enclosing blocks (6.15) or, shall be declared before its use in namespace `N` or, if `N` is a nested namespace, shall be declared before its use in one of `N`'s enclosing namespaces. [*Example:*

```
namespace A {
  namespace N {
    void f();
  }
}
void A::N::f() {
  i = 5;
  // The following scopes are searched for a declaration of i:
  // 1) outermost block scope of A::N::f, before the use of i
  // 2) scope of namespace N
  // 3) scope of namespace A
  // 4) global scope, before the definition of A::N::f
}
```

— *end example*]

1

A name used in the definition of a class `X` outside of a member function body, default argument, *exception-specification*, *brace-or-equal-initializer* of a non-static data member, or nested class definition[29] shall be declared in one of the following ways:

— before its use in class `X` or be a member of a base class of `X` (10.52), or

— if `X` is a nested class of class `Y` (9.960), before the definition of `X` in `Y`, or shall be a member of a base class of `Y` (this lookup applies in turn to `Y` 's enclosing classes, starting with the innermost enclosing class),[30] or

— if `X` is a local class (9.975) or is a nested class of a local class, before the definition of class `X` in a block enclosing the definition of class `X`, or

— if `X` is a member of namespace `N`, or is a nested class of a class that is a member of `N`, or is a local class or a nested class within a local class of a function that is a member of `N`, before the definition of class `X` in namespace `N` or in one of `N` 's enclosing namespaces.

[*Example:*

```
namespace M {
  class B { };
}
```

28) This refers to unqualified names that occur, for instance, in a type or default argument in the *parameter-declaration-clause* or used in the function body.

29) This refers to unqualified names following the class name; such a name may be used in the *base-clause* or may be used in the class definition.

30) This lookup applies whether the definition of `X` is nested within `Y`'s definition or whether `X`'s definition appears in a namespace scope enclosing `Y` 's definition (9.960).

```
namespace N {
  class Y : public M::B {
    class X {
      int a[i];
    };
  };
}

// The following scopes are searched for a declaration of i:
// 1) scope of class N::Y::X, before the use of i
// 2) scope of class N::Y, before the definition of N::Y::X
// 3) scope of N::Y's base class M::B
// 4) scope of namespace N, before the definition of N::Y
// 5) global scope, before the definition of N
```

— *end example*] [*Note:* When looking for a prior declaration of a class or function introduced by a `friend` declaration, scopes outside of the innermost enclosing namespace scope are not considered; see 7.1215.0.1. — *end note*] [*Note:* 3.760.1 further describes the restrictions on the use of names in a class definition. 9.960 further describes the restrictions on the use of names in nested class definitions. 9.975 further describes the restrictions on the use of names in local class definitions. — *end note*]

1

For the members of a class `X`, a name used in a member function body, in a default argument, in an *exception-specification*, in the *brace-or-equal-initializer* of a non-static data member (9.194), or in the definition of a class member outside of the definition of `X`, following the member's *declarator-id*[31], shall be declared in one of the following ways:

— before its use in the block in which it is used or in an enclosing block (6.15), or

— shall be a member of class `X` or be a member of a base class of `X` (10.52), or

— if `X` is a nested class of class `Y` (9.960), shall be a member of `Y`, or shall be a member of a base class of `Y` (this lookup applies in turn to `Y`'s enclosing classes, starting with the innermost enclosing class),[32] or

— if `X` is a local class (9.975) or is a nested class of a local class, before the definition of class `X` in a block enclosing the definition of class `X`, or

— if `X` is a member of namespace `N`, or is a nested class of a class that is a member of `N`, or is a local class or a nested class within a local class of a function that is a member of `N`, before the use of the name, in namespace `N` or in one of `N` 's enclosing namespaces.

[*Example:*

```
class B { };
namespace M {
  namespace N {
    class X : public B {
      void f();
    };
  }
}
```

31) That is, an unqualified name that occurs, for instance, in a type in the *parameter-declaration-clause* or in the *exception-specification*.

32) This lookup applies whether the member function is defined within the definition of class `X` or whether the member function is defined in a namespace scope enclosing `X`'s definition.

```
void M::N::X::f() {
  i = 16;
}

// The following scopes are searched for a declaration of i:
// 1) outermost block scope of M::N::X::f, before the use of i
// 2) scope of class M::N::X
// 3) scope of M::N::X's base class B
// 4) scope of namespace M::N
// 5) scope of namespace M
// 6) global scope, before the definition of M::N::X::f
```

— *end example*] [*Note:* 9.401 and 9.559 further describe the restrictions on the use of names in member function definitions. 9.960 further describes the restrictions on the use of names in the scope of nested classes. 9.975 further describes the restrictions on the use of names in local class definitions. — *end note*]

1

Name lookup for a name used in the definition of a `friend` function (11.94) defined inline in the class granting friendship shall proceed as described for lookup in member function definitions. If the `friend` function is not defined in the class granting friendship, name lookup in the `friend` function definition shall proceed as described for lookup in namespace member function definitions.

1

In a `friend` declaration naming a member function, a name used in the function declarator and not part of a *template-argument* in the *declarator-id* is first looked up in the scope of the member function's class (10.52). If it is not found, or if the name is part of a *template-argument* in the *declarator-id*, the look up is as described for unqualified names in the definition of the class granting friendship. [*Example:*

```
struct A {
  typedef int AT;
  void f1(AT);
  void f2(float);
  template <class T> void f3();
};
struct B {
  typedef char AT;
  typedef float BT;
  friend void A::f1(AT);       // parameter type is A::AT
  friend void A::f2(BT);       // parameter type is B::BT
  friend void A::f3<AT>();     // template argument is B::AT
};
```

— *end example*]

1 During the lookup for a name used as a default argument (8.545.1) in a function *parameter-declaration-clause* or used in the *expression* of a *mem-initializer* for a constructor (12.1026.1), the function parameter names are visible and hide the names of entities declared in the block, class or namespace scopes containing the function declaration. [*Note:* 8.545.1 further describes the restrictions on the use of names in default arguments. 12.1026.1 further describes the restrictions on the use of names in a *ctor-initializer*. — *end note*]

2

During the lookup of a name used in the *constant-expression* of an *enumerator-definition*, previously declared *enumerator*s of the enumeration are visible and hide the names of entities declared in the block, class, or namespace scopes containing the *enum-specifier*.

1

A name used in the definition of a `static` data member of class `X` (9.595.1) (after the *qualified-id* of the static member) is looked up as if the name was used in a member function of `X`. [*Note:* 9.595.1 further describes the restrictions on the use of names in the definition of a `static` data member. — *end note*]

1

If a variable member of a namespace is defined outside of the scope of its namespace then any name that appears in the definition of the member (after the *declarator-id*) is looked up as if the definition of the member occurred in its namespace. [*Example:*

```
namespace N {
  int i = 4;
  extern int j;
}

int i = 2;

int N::j = i;                    // N::j == 4
```

— *end example*]

1 A name used in the handler for a *function-try-block* (Clause 15) is looked up as if the name was used in the outermost block of the function definition. In particular, the function parameter names shall not be redeclared in the *exception-declaration* nor in the outermost block of a handler for the *function-try-block*. Names declared in the outermost block of the function definition are not found when looked up in the scope of a handler for the *function-try-block*. [*Note:* But function parameter names are found. — *end note*]

2 [*Note:* The rules for name lookup in template definitions are described in 14.1061. — *end note*]

3.1092.1 Argument-dependent name lookup [basic.lookup.argdep]

1

When the *postfix-expression* in a function call (5.955.1) is an *unqualified-id*, other namespaces not considered during the usual unqualified lookup (3.952.1) may be searched, and in those namespaces, namespace-scope friend function or function template declarations (11.94) not otherwise visible may be found. These modifications to the search depend on the types of the arguments (and for template template arguments, the namespace of the template argument). [*Example:*

```
namespace N {
  struct S { };
  void f(S);
}

void g() {
  N::S s;
  f(s);                          // OK: calls N::f
  (f)(s);                        // error: N::f not considered; parentheses
                                 // prevent argument-dependent lookup
}
```

— *end example*]

1

For each argument type `T` in the function call, there is a set of zero or more associated namespaces and a set of zero or more associated classes to be considered. The sets of namespaces and classes is determined entirely by the types of the function arguments (and the namespace of any template template argument).

Typedef names and *using-declaration*s used to specify the types do not contribute to this set. The sets of namespaces and classes are determined in the following way:

— If `T` is a fundamental type, its associated sets of namespaces and classes are both empty.

— If `T` is a class type (including unions), its associated classes are: the class itself; the class of which it is a member, if any; and its direct and indirect base classes. Its associated namespaces are the innermost enclosing namespaces of its associated classes. Furthermore, if `T` is a class template specialization, its associated namespaces and classes also include: the namespaces and classes associated with the types of the template arguments provided for template type parameters (excluding template template parameters); the namespaces of which any template template arguments are members; and the classes of which any member templates used as template template arguments are members. [*Note:* Non-type template arguments do not contribute to the set of associated namespaces. — *end note*]

— If `T` is an enumeration type, its associated namespace is the innermost enclosing namespace of its declaration. If it is a class member, its associated class is the member's class; else it has no associated class.

— If `T` is a pointer to `U` or an array of `U`, its associated namespaces and classes are those associated with `U`.

— If `T` is a function type, its associated namespaces and classes are those associated with the function parameter types and those associated with the return type.

— If `T` is a pointer to a member function of a class `X`, its associated namespaces and classes are those associated with the function parameter types and return type, together with those associated with `X`.

— If `T` is a pointer to a data member of class `X`, its associated namespaces and classes are those associated with the member type together with those associated with `X`.

If an associated namespace is an inline namespace (7.1149.1), its enclosing namespace is also included in the set. If an associated namespace directly contains inline namespaces, those inline namespaces are also included in the set. In addition, if the argument is the name or address of a set of overloaded functions and/or function templates, its associated classes and namespaces are the union of those associated with each of the members of the set, i.e., the classes and namespaces associated with its parameter types and return type. Additionally, if the aforementioned set of overloaded functions is named with a *template-id*, its associated classes and namespaces also include those of its type *template-argument*s and its template *template-argument*s.

1

Let X be the lookup set produced by unqualified lookup (3.952.1) and let Y be the lookup set produced by argument dependent lookup (defined as follows). If X contains

— a declaration of a class member, or

— a block-scope function declaration that is not a *using-declaration*, or

— a declaration that is neither a function or a function template

then Y is empty. Otherwise Y is the set of declarations found in the namespaces associated with the argument types as described below. The set of declarations found by the lookup of the name is the union of X and Y. [*Note:* The namespaces and classes associated with the argument types can include namespaces and classes already considered by the ordinary unqualified lookup. — *end note*] [*Example:*

```
namespace NS {
  class T { };
  void f(T);
  void g(T, int);
}
NS::T parm;
void g(NS::T, float);
int main() {
  f(parm);                      // OK: calls NS::f
  extern void g(NS::T, float);
  g(parm, 1);                   // OK: calls g(NS::T, float)
}
```

— *end example*]

1

When considering an associated namespace, the lookup is the same as the lookup performed when the associated namespace is used as a qualifier (3.1363.0.1) except that:

— Any *using-directives* in the associated namespace are ignored.

— Any namespace-scope friend functions or friend function templates declared in associated classes are visible within their respective namespaces even if they are not visible during an ordinary lookup (11.94).

— All names except those of (possibly overloaded) functions and function templates are ignored.

3.1236.1 Qualified name lookup [basic.lookup.qual]

1

The name of a class or namespace member or enumerator can be referred to after the `::` scope resolution operator (5.398) applied to a *nested-name-specifier* that denotes its class, namespace, or enumeration. If a `::` scope resolution operator in a *nested-name-specifier* is not preceded by a *decltype-specifier*, lookup of the name preceding that `::` considers only namespaces, types, and templates whose specializations are types. If the name found does not designate a namespace or a class, enumeration, or dependent type, the program is ill-formed. [*Example:*

```
class A {
public:
  static int n;
};
int main() {
  int A;
  A::n = 42;        // OK
  A b;              // ill-formed: A does not name a type
}
```

— *end example*]

1 [*Note:* Multiply qualified names, such as `N1::N2::N3::n`, can be used to refer to members of nested classes (9.960) or members of nested namespaces. — *end note*]

2

In a declaration in which the *declarator-id* is a *qualified-id*, names used before the *qualified-id* being declared are looked up in the defining namespace scope; names following the *qualified-id* are looked up in the scope of the member's class or namespace. [*Example:*

```
class X { };
class C {
  class X { };
  static const int number = 50;
  static X arr[number];
};
X C::arr[number];    // ill-formed:
                     // equivalent to: ::X C::arr[C::number];
                     // not to: C::X C::arr[C::number];
```

— *end example*]

1

A name prefixed by the unary scope operator `::` (5.398) is looked up in global scope, in the translation unit where it is used. The name shall be declared in global namespace scope or shall be a name whose declaration is visible in global scope because of a *using-directive* (3.1363.0.1). The use of `::` allows a global name to be referred to even if its identifier has been hidden (3.875.1).

1

A name prefixed by a *nested-name-specifier* that nominates an enumeration type shall represent an *enumerator* of that enumeration.

1

If a *pseudo-destructor-name* (5.1181.1) contains a *nested-name-specifier*, the *type-name*s are looked up as types in the scope designated by the *nested-name-specifier*. Similarly, in a *qualified-id* of the form:

nested-name-specifier$_{opt}$ *class-name* `::` `~` *class-name*

the second *class-name* is looked up in the same scope as the first. [*Example:*

```
struct C {
  typedef int I;
};
typedef int I1, I2;
extern int* p;
extern int* q;
p->C::I::~I();       // I is looked up in the scope of C
q->I1::~I2();        // I2 is looked up in the scope of
                     // the postfix-expression

struct A {
  ~A();
};
typedef A AB;
int main() {
  AB* p;
  p->AB::~AB();      // explicitly calls the destructor for A
}
```

— *end example*] [*Note:* 3.1423.1 describes how name lookup proceeds after the `.` and `->` operators. — *end note*]

3.1282.0.1 Class members [class.qual]

1

If the *nested-name-specifier* of a *qualified-id* nominates a class, the name specified after the *nested-name-specifier* is looked up in the scope of the class (10.52), except for the cases listed below. The name shall

represent one or more members of that class or of one of its base classes (Clause 10). [*Note:* A class member can be referred to using a *qualified-id* at any point in its potential scope (3.760.1). — *end note*] The exceptions to the name lookup rule above are the following:

— a destructor name is looked up as specified in 3.1236.1;

— a *conversion-type-id* of a *conversion-function-id* is looked up in the same manner as a *conversion-type-id* in a class member access (see 3.1423.1);

— the names in a *template-argument* of a *template-id* are looked up in the context in which the entire *postfix-expression* occurs.

— the lookup for a name specified in a *using-declaration* (7.1311.1) also finds class or enumeration names hidden within the same scope (3.875.1).

1

In a lookup in which function names are not ignored[33] and the *nested-name-specifier* nominates a class `C`:

— if the name specified after the *nested-name-specifier*, when looked up in `C`, is the injected-class-name of `C` (Clause 9), or

— in a *using-declaration* (7.1311.1) that is a *member-declaration*, if the name specified after the *nested-name-specifier* is the same as the *identifier* or the *simple-template-id*'s *template-name* in the last component of the *nested-name-specifier*,

the name is instead considered to name the constructor of class `C`. [*Note:* For example, the constructor is not an acceptable lookup result in an *elaborated-type-specifier* so the constructor would not be used in place of the injected-class-name. — *end note*] Such a constructor name shall be used only in the *declarator-id* of a declaration that names a constructor or in a *using-declaration*. [*Example:*

```
struct A { A(); };
struct B: public A { B(); };

A::A() { }
B::B() { }

B::A ba;            // object of type A
A::A a;             // error, A::A is not a type name
struct A::A a2;     // object of type A
```

— *end example*]

1

A class member name hidden by a name in a nested declarative region or by the name of a derived class member can still be found if qualified by the name of its class followed by the `::` operator.

3.1363.0.1 Namespace members [namespace.qual]

1

If the *nested-name-specifier* of a *qualified-id* nominates a namespace, the name specified after the *nested-name-specifier* is looked up in the scope of the namespace. If a *qualified-id* starts with `::`, the name after the `::` is looked up in the global namespace. In either case, the names in a *template-argument* of a *template-id* are looked up in the context in which the entire *postfix-expression* occurs.

33) Lookups in which function names are ignored include names appearing in a *nested-name-specifier*, an *elaborated-type-specifier*, or a *base-specifier*.

1

For a namespace `X` and name `m`, the namespace-qualified lookup set $S(X, m)$ is defined as follows: Let $S'(X, m)$ be the set of all declarations of `m` in `X` and the inline namespace set of `X` (7.1149.1). If $S'(X, m)$ is not empty, $S(X, m)$ is $S'(X, m)$; otherwise, $S(X, m)$ is the union of $S(N_i, m)$ for all namespaces N_i nominated by *using-directives* in `X` and its inline namespace set.

1

Given `X::m` (where `X` is a user-declared namespace), or given `::m` (where X is the global namespace), if $S(X, m)$ is the empty set, the program is ill-formed. Otherwise, if $S(X, m)$ has exactly one member, or if the context of the reference is a *using-declaration* (7.1311.1), $S(X, m)$ is the required set of declarations of `m`. Otherwise if the use of `m` is not one that allows a unique declaration to be chosen from $S(X, m)$, the program is ill-formed. [*Example:*

```
int x;
namespace Y {
  void f(float);
  void h(int);
}

namespace Z {
  void h(double);
}

namespace A {
  using namespace Y;
  void f(int);
  void g(int);
  int i;
}

namespace B {
  using namespace Z;
  void f(char);
  int i;
}

namespace AB {
  using namespace A;
  using namespace B;
  void g();
}

void h()
{
  AB::g();          // g is declared directly in AB,
                    // therefore S is { AB::g() } and AB::g() is chosen
  AB::f(1);         // f is not declared directly in AB so the rules are
                    // applied recursively to A and B;
                    // namespace Y is not searched and Y::f(float)
                    // is not considered;
                    // S is { A::f(int), B::f(char) } and overload
                    // resolution chooses A::f(int)
  AB::f('c');       // as above but resolution chooses B::f(char)

  AB::x++;          // x is not declared directly in AB, and
```

```
                        // is not declared in A or B , so the rules are
                        // applied recursively to Y and Z,
                        // S is { } so the program is ill-formed
  AB::i++;              // i is not declared directly in AB so the rules are
                        // applied recursively to A and B,
                        // S is { A::i , B::i } so the use is ambiguous
                        // and the program is ill-formed
  AB::h(16.8);          // h is not declared directly in AB and
                        // not declared directly in A or B so the rules are
                        // applied recursively to Y and Z,
                        // S is { Y::h(int), Z::h(double) } and overload
                        // resolution chooses Z::h(double)
}
```

1 The same declaration found more than once is not an ambiguity (because it is still a unique declaration). For example:

```
namespace A {
  int a;
}

namespace B {
  using namespace A;
}

namespace C {
  using namespace A;
}

namespace BC {
  using namespace B;
  using namespace C;
}

void f()
{
  BC::a++;            // OK: S is { A::a, A::a }
}

namespace D {
  using A::a;
}

namespace BD {
  using namespace B;
  using namespace D;
}

void g()
{
  BD::a++;            // OK: S is {  A::a,  A::a }
}
```

2 Because each referenced namespace is searched at most once, the following is well-defined:

```
namespace B {
```

```
  int b;
}

namespace A {
  using namespace B;
  int a;
}

namespace B {
  using namespace A;
}

void f()
{
  A::a++;          // OK: a declared directly in A, S is {A::a}
  B::a++;          // OK: both A and B searched (once), S is {A::a}
  A::b++;          // OK: both A and B searched (once), S is {B::b}
  B::b++;          // OK: b declared directly in B, S is {B::b}
}
```

— *end example*]

3 During the lookup of a qualified namespace member name, if the lookup finds more than one declaration of the member, and if one declaration introduces a class name or enumeration name and the other declarations either introduce the same variable, the same enumerator or a set of functions, the non-type name hides the class or enumeration name if and only if the declarations are from the same namespace; otherwise (the declarations are from different namespaces), the program is ill-formed. [*Example:*

```
namespace A {
  struct x { };
  int x;
  int y;
}

namespace B {
  struct y { };
}

namespace C {
  using namespace A;
  using namespace B;
  int i = C::x;     // OK, A::x (of type int )
  int j = C::y;     // ambiguous, A::y or B::y
}
```

— *end example*]

4

In a declaration for a namespace member in which the *declarator-id* is a *qualified-id*, given that the *qualified-id* for the namespace member has the form

nested-name-specifier unqualified-id

the *unqualified-id* shall name a member of the namespace designated by the *nested-name-specifier* or of an element of the inline namespace set (7.1149.1) of that namespace. [*Example:*

```
namespace A {
```

```
  namespace B {
    void f1(int);
  }
  using namespace B;
}
void A::f1(int){ }  // ill-formed, f1 is not a member of A
```

— *end example*] However, in such namespace member declarations, the *nested-name-specifier* may rely on *using-directives* to implicitly provide the initial part of the *nested-name-specifier*. [*Example:*

```
namespace A {
  namespace B {
    void f1(int);
  }
}

namespace C {
  namespace D {
    void f1(int);
  }
}

using namespace A;
using namespace C::D;
void B::f1(int){ }  // OK, defines A::B::f1(int)
```

— *end example*]

3.1395.1 Elaborated type specifiers [basic.lookup.elab]

1

An *elaborated-type-specifier* (7.808.0.1) may be used to refer to a previously declared *class-name* or *enum-name* even though the name has been hidden by a non-type declaration (3.875.1).

1

If the *elaborated-type-specifier* has no *nested-name-specifier*, and unless the *elaborated-type-specifier* appears in a declaration with the following form:

class-key attribute-specifier-seq$_{opt}$ *identifier* ;

the *identifier* is looked up according to 3.952.1 but ignoring any non-type names that have been declared. If the *elaborated-type-specifier* is introduced by the `enum` keyword and this lookup does not find a previously declared *type-name*, the *elaborated-type-specifier* is ill-formed. If the *elaborated-type-specifier* is introduced by the *class-key* and this lookup does not find a previously declared *type-name*, or if the *elaborated-type-specifier* appears in a declaration with the form:

class-key attribute-specifier-seq$_{opt}$ *identifier* ;

the *elaborated-type-specifier* is a declaration that introduces the *class-name* as described in 3.536.1.

1 If the *elaborated-type-specifier* has a *nested-name-specifier*, qualified name lookup is performed, as described in 3.1236.1, but ignoring any non-type names that have been declared. If the name lookup does not find a previously declared *type-name*, the *elaborated-type-specifier* is ill-formed. [*Example:*

```
struct Node {
  struct Node* Next;          // OK: Refers to Node at global scope
  struct Data* Data;          // OK: Declares type Data
                              // at global scope and member Data
```

```
};

struct Data {
  struct Node* Node;            // OK: Refers to Node at global scope
  friend struct ::Glob;         // error: Glob is not declared
                                // cannot introduce a qualified type (7.808.0.1)
  friend struct Glob;           // OK: Refers to (as yet) undeclared Glob
                                // at global scope.
  /* ... */
};

struct Base {
  struct Data;                  // OK: Declares nested Data
  struct ::Data*     thatData;  // OK: Refers to ::Data
  struct Base::Data* thisData;  // OK: Refers to nested Data
  friend class ::Data;          // OK: global Data is a friend
  friend class Data;            // OK: nested Data is a friend
  struct Data { /* ...  */ };   // Defines nested Data
};

struct Data;                    // OK: Redeclares Data at global scope
struct ::Data;                  // error: cannot introduce a qualified type (7.808.0.1)
struct Base::Data;              // error: cannot introduce a qualified type (7.808.0.1)
struct Base::Datum;             // error: Datum undefined
struct Base::Data* pBase;       // OK: refers to nested Data
```

— *end example*]

3.1423.1 Class member access [basic.lookup.classref]

1

In a class member access expression (5.1194.1), if the `.` or `->` token is immediately followed by an *identifier* followed by a `<`, the identifier must be looked up to determine whether the `<` is the beginning of a template argument list (14.224) or a less-than operator. The identifier is first looked up in the class of the object expression. If the identifier is not found, it is then looked up in the context of the entire *postfix-expression* and shall name a class template.

1

If the *id-expression* in a class member access (5.1194.1) is an *unqualified-id*, and the type of the object expression is of a class type `C`, the *unqualified-id* is looked up in the scope of class `C`. For a pseudo-destructor call (5.1181.1), the *unqualified-id* is looked up in the context of the complete *postfix-expression*.

1

If the *unqualified-id* is `~`*type-name*, the *type-name* is looked up in the context of the entire *postfix-expression*. If the type `T` of the object expression is of a class type `C`, the *type-name* is also looked up in the scope of class `C`. At least one of the lookups shall find a name that refers to (possibly cv-qualified) `T`. [*Example:*

```
struct A { };

struct B {
  struct A { };
  void f(::A* a);
};

void B::f(::A* a) {
```

```
  a->~A();                    // OK: lookup in *a finds the injected-class-name
}
```

— *end example*]

1

If the *id-expression* in a class member access is a *qualified-id* of the form

```
class-name-or-namespace-name::...
```

the *class-name-or-namespace-name* following the `.` or `->` operator is first looked up in the class of the object expression and the name, if found, is used. Otherwise it is looked up in the context of the entire *postfix-expression*. [*Note:* See 3.1236.1, which describes the lookup of a name before `::`, which will only find a type or namespace name. — *end note*]

1 If the *qualified-id* has the form

```
::class-name-or-namespace-name::...
```

the *class-name-or-namespace-name* is looked up in global scope as a *class-name* or *namespace-name*.

2 If the *nested-name-specifier* contains a *simple-template-id* (14.224), the names in its *template-arguments* are looked up in the context in which the entire *postfix-expression* occurs.

3

If the *id-expression* is a *conversion-function-id*, its *conversion-type-id* is first looked up in the class of the object expression and the name, if found, is used. Otherwise it is looked up in the context of the entire *postfix-expression*. In each of these lookups, only names that denote types or templates whose specializations are types are considered. [*Example:*

```
struct A { };
namespace N {
  struct A {
    void g() { }
    template <class T> operator T();
  };
}

int main() {
  N::A a;
  a.operator A();             // calls N::A::operator N::A
}
```

— *end example*]

3.1457.1 Using-directives and namespace aliases [basic.lookup.udir]

1

In a *using-directive* or *namespace-alias-definition*, during the lookup for a *namespace-name* or for a name in a *nested-name-specifier* only namespace names are considered.

3.1459 Program and linkage [basic.link]

1

A *program* consists of one or more *translation units* (Clause 2) linked together. A translation unit consists of a sequence of declarations.

translation-unit:
  declaration-seq$_{opt}$

1

A name is said to have *linkage* when it might denote the same object, reference, function, type, template, namespace or value as a name introduced by a declaration in another scope:

— When a name has *external linkage* , the entity it denotes can be referred to by names from scopes of other translation units or from other scopes of the same translation unit.

— When a name has *internal linkage* , the entity it denotes can be referred to by names from other scopes in the same translation unit.

— When a name has *no linkage* , the entity it denotes cannot be referred to by names from other scopes.

1

A name having namespace scope (3.736.1) has internal linkage if it is the name of

— a variable, function or function template that is explicitly declared `static`; or,

— a non-volatile variable that is explicitly declared `const` or `constexpr` and neither explicitly declared `extern` nor previously declared to have external linkage; or

— a data member of an anonymous union.

1

An unnamed namespace or a namespace declared directly or indirectly within an unnamed namespace has internal linkage. All other namespaces have external linkage. A name having namespace scope that has not been given internal linkage above has the same linkage as the enclosing namespace if it is the name of

— a variable; or

— a function; or

— a named class (Clause 9), or an unnamed class defined in a typedef declaration in which the class has the typedef name for linkage purposes (7.235.1); or

— a named enumeration (7.1001), or an unnamed enumeration defined in a typedef declaration in which the enumeration has the typedef name for linkage purposes (7.235.1); or

— an enumerator belonging to an enumeration with linkage; or

— a template.

1

In addition, a member function, static data member, a named class or enumeration of class scope, or an unnamed class or enumeration defined in a class-scope typedef declaration such that the class or enumeration has the typedef name for linkage purposes (7.235.1), has external linkage if the name of the class has external linkage.

1

The name of a function declared in block scope and the name of a variable declared by a block scope `extern` declaration have linkage. If there is a visible declaration of an entity with linkage having the same name and type, ignoring entities declared outside the innermost enclosing namespace scope, the block scope declaration declares that same entity and receives the linkage of the previous declaration. If there is more than one such matching entity, the program is ill-formed. Otherwise, if no matching entity is found, the block scope entity receives external linkage.[*Example:*

```
static void f();
static int i = 0;              // #1
void g() {
  extern void f();             // internal linkage
  int i;                       // #2 i has no linkage
  {
    extern void f();           // internal linkage
    extern int i;              // #3 external linkage
  }
}
```

There are three objects named `i` in this program. The object with internal linkage introduced by the declaration in global scope (line `#1`), the object with automatic storage duration and no linkage introduced by the declaration on line `#2`, and the object with static storage duration and external linkage introduced by the declaration on line `#3`. — *end example*]

1

When a block scope declaration of an entity with linkage is not found to refer to some other declaration, then that entity is a member of the innermost enclosing namespace. However such a declaration does not introduce the member name in its namespace scope. [*Example:*

```
namespace X {
  void p() {
    q();                       // error: q not yet declared
    extern void q();           // q is a member of namespace X
  }

  void middle() {
    q();                       // error: q not yet declared
  }

  void q() { /* ...  */ }      // definition of X::q
}

void q() { /* ...  */ }        // some other, unrelated q
```

— *end example*]

1

Names not covered by these rules have no linkage. Moreover, except as noted, a name declared at block scope (3.694.1) has no linkage. A type is said to have linkage if and only if:

— it is a class or enumeration type that is named (or has a name for linkage purposes (7.235.1)) and the name has linkage; or

— it is an unnamed class or enumeration member of a class with linkage; or

— it is a specialization of a class template (Clause 14)[34]; or

— it is a fundamental type (3.3151.1); or

— it is a compound type (3.3568.1) other than a class or enumeration, compounded exclusively from types that have linkage; or

34) A class template always has external linkage, and the requirements of 14.349.1 and 14.364.1 ensure that the template arguments will also have appropriate linkage.

— it is a cv-qualified (3.3640.1) version of a type that has linkage.

A type without linkage shall not be used as the type of a variable or function with external linkage unless

— the entity has C language linkage (7.1477), or

— the entity is declared within an unnamed namespace (7.1149.1), or

— the entity is not odr-used (3.91) or is defined in the same translation unit.

[*Note:* In other words, a type without linkage contains a class or enumeration that cannot be named outside its translation unit. An entity with external linkage declared using such a type could not correspond to any other entity in another translation unit of the program and thus must be defined in the translation unit if it is odr-used. Also note that classes with linkage may contain members whose types do not have linkage, and that typedef names are ignored in the determination of whether a type has linkage. — *end note*]

[*Example:*

```
template <class T> struct B {
  void g(T) { }
  void h(T);
  friend void i(B, T) { }
};

void f() {
  struct A { int x; };  // no linkage
  A a = { 1 };
  B<A> ba;              // declares B<A>::g(A) and B<A>::h(A)
  ba.g(a);              // OK
  ba.h(a);              // error: B<A>::h(A) not defined in the translation unit
  i(ba, a);             // OK
}
```

— *end example*]

1

Two names that are the same (Clause 3) and that are declared in different scopes shall denote the same variable, function, type, enumerator, template or namespace if

— both names have external linkage or else both names have internal linkage and are declared in the same translation unit; and

— both names refer to members of the same namespace or to members, not by inheritance, of the same class; and

— when both names denote functions, the parameter-type-lists of the functions (8.360.1) are identical; and

— when both names denote function templates, the signatures (14.975.0.1) are the same.

1

After all adjustments of types (during which typedefs (7.235.1) are replaced by their definitions), the types specified by all declarations referring to a given variable or function shall be identical, except that declarations for an array object can specify array types that differ by the presence or absence of a major array bound (8.251.1). A violation of this rule on type identity does not require a diagnostic.

1

[*Note:* Linkage to non-C++ declarations can be achieved using a *linkage-specification* (7.1477). — *end note*]

3.1721 Start and termination [basic.start]

3.1721.1 Main function [basic.start.main]

1

A program shall contain a global function called `main`, which is the designated start of the program. It is implementation-defined whether a program in a freestanding environment is required to define a `main` function. [*Note:* In a freestanding environment, start-up and termination is implementation-defined; start-up contains the execution of constructors for objects of namespace scope with static storage duration; termination contains the execution of destructors for objects with static storage duration. — *end note*]

1

An implementation shall not predefine the `main` function. This function shall not be overloaded. It shall have a declared return type of type `int`, but otherwise its type is implementation-defined. An implementation shall allow both

— a function of `()` returning `int` and

— a function of `(int`, pointer to pointer to `char)` returning `int`

as the type of `main` (8.360.1). In the latter form, for purposes of exposition, the first function parameter is called `argc` and the second function parameter is called `argv`, where `argc` shall be the number of arguments passed to the program from the environment in which the program is run. If `argc` is nonzero these arguments shall be supplied in `argv[0]` through `argv[argc-1]` as pointers to the initial characters of null-terminated multibyte strings (NTMBS s) (17.26.0.0.0.1) and `argv[0]` shall be the pointer to the initial character of a NTMBS that represents the name used to invoke the program or `""`. The value of `argc` shall be non-negative. The value of `argv[argc]` shall be 0. [*Note:* It is recommended that any further (optional) parameters be added after `argv`. — *end note*]

1

The function `main` shall not be used within a program. The linkage (3.1459) of `main` is implementation-defined. A program that defines `main` as deleted or that declares `main` to be `inline`, `static`, or `constexpr` is ill-formed. The name `main` is not otherwise reserved. [*Example:* member functions, classes, and enumerations can be called `main`, as can entities in other namespaces. — *end example*]

1

Terminating the program without leaving the current block (e.g., by calling the function `std::exit(int)` (18.127)) does not destroy any objects with automatic storage duration (12.807). If `std::exit` is called to end a program during the destruction of an object with static or thread storage duration, the program has undefined behavior.

1

A return statement in `main` has the effect of leaving the main function (destroying any objects with automatic storage duration) and calling `std::exit` with the return value as the argument. If control reaches the end of `main` without encountering a `return` statement, the effect is that of executing

```
return 0;
```

3.1937.1 Initialization of non-local variables [basic.start.init]

1

There are two broad classes of named non-local variables: those with static storage duration (3.2242.1) and those with thread storage duration (3.2291.1). Non-local variables with static storage duration are initialized as a consequence of program initiation. Non-local variables with thread storage duration are initialized as a consequence of thread execution. Within each of these phases of initiation, initialization occurs as follows.

1

Variables with static storage duration (3.2242.1) or thread storage duration (3.2291.1) shall be zero-initialized (8.821) before any other initialization takes place. A *constant initializer* for an object `o` is an expression that is a constant expression, except that it may also invoke `constexpr` constructors for `o` and its subobjects even if those objects are of non-literal class types [*Note:* such a class may have a non-trivial destructor — *end note*]. *Constant initialization* is performed:

— if each full-expression (including implicit conversions) that appears in the initializer of a reference with static or thread storage duration is a constant expression (5.3643) and the reference is bound to an lvalue designating an object with static storage duration, to a temporary (see 12.306), or to a function;

— if an object with static or thread storage duration is initialized by a constructor call, and if the initialization full-expression is a constant initializer for the object;

— if an object with static or thread storage duration is not initialized by a constructor call and if either the object is value-initialized or every full-expression that appears in its initializer is a constant expression.

Together, zero-initialization and constant initialization are called *static initialization*; all other initialization is *dynamic initialization*. Static initialization shall be performed before any dynamic initialization takes place. Dynamic initialization of a non-local variable with static storage duration is either ordered or unordered. Definitions of explicitly specialized class template static data members have ordered initialization. Other class template static data members (i.e., implicitly or explicitly instantiated specializations) have unordered initialization. Other non-local variables with static storage duration have ordered initialization. Variables with ordered initialization defined within a single translation unit shall be initialized in the order of their definitions in the translation unit. If a program starts a thread (30.25), the subsequent initialization of a variable is unsequenced with respect to the initialization of a variable defined in a different translation unit. Otherwise, the initialization of a variable is indeterminately sequenced with respect to the initialization of a variable defined in a different translation unit. If a program starts a thread, the subsequent unordered initialization of a variable is unsequenced with respect to every other dynamic initialization. Otherwise, the unordered initialization of a variable is indeterminately sequenced with respect to every other dynamic initialization. [*Note:* This definition permits initialization of a sequence of ordered variables concurrently with another sequence. — *end note*] [*Note:* The initialization of local static variables is described in 6.419. — *end note*]

1

An implementation is permitted to perform the initialization of a non-local variable with static storage duration as a static initialization even if such initialization is not required to be done statically, provided that

— the dynamic version of the initialization does not change the value of any other object of namespace scope prior to its initialization, and

— the static version of the initialization produces the same value in the initialized variable as would be produced by the dynamic initialization if all variables not required to be initialized statically were initialized dynamically.

[*Note:* As a consequence, if the initialization of an object `obj1` refers to an object `obj2` of namespace scope potentially requiring dynamic initialization and defined later in the same translation unit, it is unspecified

whether the value of `obj2` used will be the value of the fully initialized `obj2` (because `obj2` was statically initialized) or will be the value of `obj2` merely zero-initialized. For example,

```
inline double fd() { return 1.0; }
extern double d1;
double d2 = d1;      // unspecified:
                     // may be statically initialized to 0.0 or
                     // dynamically initialized to 0.0 if d1 is
                     // dynamically initialized, or 1.0 otherwise
double d1 = fd();    // may be initialized statically or dynamically to 1.0
```

— *end note*]

1

It is implementation-defined whether the dynamic initialization of a non-local variable with static storage duration is done before the first statement of `main`. If the initialization is deferred to some point in time after the first statement of `main`, it shall occur before the first odr-use (3.91) of any function or variable defined in the same translation unit as the variable to be initialized.[35] [*Example:*

```
// - File 1 -
#include "a.h"
#include "b.h"
B b;
A::A(){
  b.Use();
}

// - File 2 -
#include "a.h"
A a;

// - File 3 -
#include "a.h"
#include "b.h"
extern A a;
extern B b;

int main() {
  a.Use();
  b.Use();
}
```

It is implementation-defined whether either `a` or `b` is initialized before `main` is entered or whether the initializations are delayed until `a` is first odr-used in `main`. In particular, if `a` is initialized before `main` is entered, it is not guaranteed that `b` will be initialized before it is odr-used by the initialization of `a`, that is, before `A::A` is called. If, however, `a` is initialized at some point after the first statement of `main`, `b` will be initialized prior to its use in `A::A`. — *end example*]

1

It is implementation-defined whether the dynamic initialization of a non-local variable with static or thread storage duration is done before the first statement of the initial function of the thread. If the initialization is deferred to some point in time after the first statement of the initial function of the thread, it shall occur

35) A non-local variable with static storage duration having initialization with side-effects must be initialized even if it is not odr-used (3.91, 3.2242.1).

before the first odr-use (3.91) of any variable with thread storage duration defined in the same translation unit as the variable to be initialized.

1

If the initialization of a non-local variable with static or thread storage duration exits via an exception, `std::terminate` is called (15.308.1).

3.2200.1 Termination [basic.start.term]

1

Destructors (12.807) for initialized objects (that is, objects whose lifetime (3.2563) has begun) with static storage duration are called as a result of returning from `main` and as a result of calling `std::exit` (18.127). Destructors for initialized objects with thread storage duration within a given thread are called as a result of returning from the initial function of that thread and as a result of that thread calling `std::exit`. The completions of the destructors for all initialized objects with thread storage duration within that thread are sequenced before the initiation of the destructors of any object with static storage duration. If the completion of the constructor or dynamic initialization of an object with thread storage duration is sequenced before that of another, the completion of the destructor of the second is sequenced before the initiation of the destructor of the first. If the completion of the constructor or dynamic initialization of an object with static storage duration is sequenced before that of another, the completion of the destructor of the second is sequenced before the initiation of the destructor of the first. [*Note:* This definition permits concurrent destruction. — *end note*] If an object is initialized statically, the object is destroyed in the same order as if the object was dynamically initialized. For an object of array or class type, all subobjects of that object are destroyed before any block-scope object with static storage duration initialized during the construction of the subobjects is destroyed. If the destruction of an object with static or thread storage duration exits via an exception, `std::terminate` is called (15.308.1).

1

If a function contains a block-scope object of static or thread storage duration that has been destroyed and the function is called during the destruction of an object with static or thread storage duration, the program has undefined behavior if the flow of control passes through the definition of the previously destroyed block-scope object. Likewise, the behavior is undefined if the block-scope object is used indirectly (i.e., through a pointer) after its destruction.

1

If the completion of the initialization of an object with static storage duration is sequenced before a call to `std::atexit` (see `<cstdlib>`, 18.127), the call to the function passed to `std::atexit` is sequenced before the call to the destructor for the object. If a call to `std::atexit` is sequenced before the completion of the initialization of an object with static storage duration, the call to the destructor for the object is sequenced before the call to the function passed to `std::atexit`. If a call to `std::atexit` is sequenced before another call to `std::atexit`, the call to the function passed to the second `std::atexit` call is sequenced before the call to the function passed to the first `std::atexit` call.

1

If there is a use of a standard library object or function not permitted within signal handlers (18.295) that does not happen before (1.859) completion of destruction of objects with static storage duration and execution of `std::atexit` registered functions (18.127), the program has undefined behavior. [*Note:* If there is a use of an object with static storage duration that does not happen before the object's destruction, the program has undefined behavior. Terminating every thread before a call to `std::exit` or the exit from `main` is sufficient, but not necessary, to satisfy these requirements. These requirements permit thread managers as static-storage-duration objects. — *end note*]

1 Calling the function `std::abort()` declared in `<cstdlib>` terminates the program without executing any

destructors and without calling the functions passed to `std::atexit()` or `std::at_quick_exit()`.

3.2235 Storage duration [basic.stc]

1

Storage duration is the property of an object that defines the minimum potential lifetime of the storage containing the object. The storage duration is determined by the construct used to create the object and is one of the following:

— static storage duration

— thread storage duration

— automatic storage duration

— dynamic storage duration

1

Static, thread, and automatic storage durations are associated with objects introduced by declarations (3.43) and implicitly created by the implementation (12.306). The dynamic storage duration is associated with objects created with `operator new` (5.2221.1).

1

The storage duration categories apply to references as well. The lifetime of a reference is its storage duration.

3.2242.1 Static storage duration [basic.stc.static]

1

All variables which do not have dynamic storage duration, do not have thread storage duration, and are not local have *static storage duration*. The storage for these entities shall last for the duration of the program (3.1937.1, 3.2200.1).

1

If a variable with static storage duration has initialization or a destructor with side effects, it shall not be eliminated even if it appears to be unused, except that a class object or its copy/move may be eliminated as specified in 12.1577.

1

The keyword `static` can be used to declare a local variable with static storage duration. [*Note:* 6.419 describes the initialization of local `static` variables; 3.2200.1 describes the destruction of local `static` variables. — *end note*]

1

The keyword `static` applied to a class data member in a class definition gives the data member static storage duration.

3.2291.1 Thread storage duration [basic.stc.thread]

1

All variables declared with the `thread_local` keyword have *thread storage duration*. The storage for these entities shall last for the duration of the thread in which they are created. There is a distinct object or reference per thread, and use of the declared name refers to the entity associated with the current thread.

1

A variable with thread storage duration shall be initialized before its first odr-use (3.91) and, if constructed, shall be destroyed on thread exit.

3.2301.1 Automatic storage duration [basic.stc.auto]

1

Block-scope variables explicitly declared `register` or not explicitly declared `static` or `extern` have *automatic storage duration*. The storage for these entities lasts until the block in which they are created exits.

1 [*Note:* These variables are initialized and destroyed as described in 6.419. — *end note*]

2

If a variable with automatic storage duration has initialization or a destructor with side effects, it shall not be destroyed before the end of its block, nor shall it be eliminated as an optimization even if it appears to be unused, except that a class object or its copy/move may be eliminated as specified in 12.1577.

3.2331.1 Dynamic storage duration [basic.stc.dynamic]

1

Objects can be created dynamically during program execution (1.430), using *new-expression*s (5.2221.1), and destroyed using *delete-expression*s (5.2455.1). A C++ implementation provides access to, and management of, dynamic storage via the global *allocation functions* `operator new` and `operator new[]` and the global *deallocation functions* `operator delete` and `operator delete[]`.

1

The library provides default definitions for the global allocation and deallocation functions. Some global allocation and deallocation functions are replaceable (18.159.1). A C++ program shall provide at most one definition of a replaceable allocation or deallocation function. Any such function definition replaces the default version provided in the library (17.428.0.3). The following allocation and deallocation functions (18.159) are implicitly declared in global scope in each translation unit of a program.

```
void* operator new(std::size_t);
void* operator new[](std::size_t);
void operator delete(void*) noexcept;
void operator delete[](void*) noexcept;
void operator delete(void*, std::size_t) noexcept;
void operator delete[](void*, std::size_t) noexcept;
```

These implicit declarations introduce only the function names `operator new`, `operator new[]`, `operator delete`, and `operator delete[]`. [*Note:* The implicit declarations do not introduce the names `std`, `std::size_t`, or any other names that the library uses to declare these names. Thus, a *new-expression*, *delete-expression* or function call that refers to one of these functions without including the header `<new>` is well-formed. However, referring to `std` or `std::size_t` is ill-formed unless the name has been declared by including the appropriate header. — *end note*] Allocation and/or deallocation functions can also be declared and defined for any class (12.988).

1

Any allocation and/or deallocation functions defined in a C++ program, including the default versions in the library, shall conform to the semantics specified in 3.2361.0.1 and 3.2428.0.1.

3.2361.0.1 Allocation functions [basic.stc.dynamic.allocation]

1

An allocation function shall be a class member function or a global function; a program is ill-formed if an allocation function is declared in a namespace scope other than global scope or declared static in global scope. The return type shall be `void*`. The first parameter shall have type `std::size_t` (18.2). The first parameter shall not have an associated default argument (8.545.1). The value of the first parameter shall be

interpreted as the requested size of the allocation. An allocation function can be a function template. Such a template shall declare its return type and first parameter as specified above (that is, template parameter types shall not be used in the return type and first parameter type). Template allocation functions shall have two or more parameters.

1

The allocation function attempts to allocate the requested amount of storage. If it is successful, it shall return the address of the start of a block of storage whose length in bytes shall be at least as large as the requested size. There are no constraints on the contents of the allocated storage on return from the allocation function. The order, contiguity, and initial value of storage allocated by successive calls to an allocation function are unspecified. The pointer returned shall be suitably aligned so that it can be converted to a pointer of any complete object type with a fundamental alignment requirement (3.3924) and then used to access the object or array in the storage allocated (until the storage is explicitly deallocated by a call to a corresponding deallocation function). Even if the size of the space requested is zero, the request can fail. If the request succeeds, the value returned shall be a non-null pointer value (4.579) `p0` different from any previously returned value `p1`, unless that value `p1` was subsequently passed to an `operator delete`. The effect of indirecting through a pointer returned as a request for zero size is undefined.[36]

1

An allocation function that fails to allocate storage can invoke the currently installed new-handler function (18.226.0.1), if any. [*Note:* A program-supplied allocation function can obtain the address of the currently installed `new_handler` using the `std::get_new_handler` function (18.226.0.2). — *end note*] If an allocation function declared with a non-throwing *exception-specification* (15.228) fails to allocate storage, it shall return a null pointer. Any other allocation function that fails to allocate storage shall indicate failure only by throwing an exception (15.7) of a type that would match a handler (15.146) of type `std::bad_alloc` (18.223.1.1).

1

A global allocation function is only called as the result of a new expression (5.2221.1), or called directly using the function call syntax (5.955.1), or called indirectly through calls to the functions in the C++ standard library. [*Note:* In particular, a global allocation function is not called to allocate storage for objects with static storage duration (3.2242.1), for objects or references with thread storage duration (3.2291.1), for objects of type `std::type_info` (5.1393.1), or for an exception object (15.7). — *end note*]

3.2428.0.1 Deallocation functions [basic.stc.dynamic.deallocation]

1

Deallocation functions shall be class member functions or global functions; a program is ill-formed if deallocation functions are declared in a namespace scope other than global scope or declared static in global scope.

1

Each deallocation function shall return `void` and its first parameter shall be `void*`. A deallocation function can have more than one parameter. The global `operator delete` with exactly one parameter is a usual (non-placement) deallocation function. The global `operator delete` with exactly two parameters, the second of which has type `std::size_t`, is a usual deallocation function. Similarly, the global `operator delete[]` with exactly one parameter is a usual deallocation function. The global `operator delete[]` with exactly two parameters, the second of which has type `std::size_t`, is a usual deallocation function.[37] If a class `T`

36) The intent is to have `operator new()` implementable by calling `std::malloc()` or `std::calloc()`, so the rules are substantially the same. C++ differs from C in requiring a zero request to return a non-null pointer.

37) This deallocation function precludes use of an allocation function `void operator new(std::size_t, std::size_t)` as a placement allocation function (C.3.2).

has a member deallocation function named `operator delete` with exactly one parameter, then that function is a usual deallocation function. If class `T` does not declare such an `operator delete` but does declare a member deallocation function named `operator delete` with exactly two parameters, the second of which has type `std::size_t`, then this function is a usual deallocation function. Similarly, if a class `T` has a member deallocation function named `operator delete[]` with exactly one parameter, then that function is a usual (non-placement) deallocation function. If class `T` does not declare such an `operator delete[]` but does declare a member deallocation function named `operator delete[]` with exactly two parameters, the second of which has type `std::size_t`, then this function is a usual deallocation function. A deallocation function can be an instance of a function template. Neither the first parameter nor the return type shall depend on a template parameter. [*Note:* That is, a deallocation function template shall have a first parameter of type `void*` and a return type of `void` (as specified above). — *end note*] A deallocation function template shall have two or more function parameters. A template instance is never a usual deallocation function, regardless of its signature.

1

If a deallocation function terminates by throwing an exception, the behavior is undefined. The value of the first argument supplied to a deallocation function may be a null pointer value; if so, and if the deallocation function is one supplied in the standard library, the call has no effect. Otherwise, the behavior is undefined if the value supplied to `operator delete(void*)` in the standard library is not one of the values returned by a previous invocation of either `operator new(std::size_t)` or `operator new(std::size_t, const std::nothrow_t&)` in the standard library, and the behavior is undefined if the value supplied to `operator delete[](void*)` in the standard library is not one of the values returned by a previous invocation of either `operator new[](std::size_t)` or `operator new[](std::size_t, const std::nothrow_t&)` in the standard library.

1

If the argument given to a deallocation function in the standard library is a pointer that is not the null pointer value (4.579), the deallocation function shall deallocate the storage referenced by the pointer, rendering invalid all pointers referring to any part of the deallocated storage. Indirection through an invalid pointer value and passing an invalid pointer value to a deallocation function have undefined behavior. Any other use of an invalid pointer value has implementation-defined behavior.[38]

3.2532.0.1 Safely-derived pointers [basic.stc.dynamic.safety]

1

A *traceable pointer object* is

— an object of an object pointer type (3.3568.1), or

— an object of an integral type that is at least as large as `std::intptr_t`, or

— a sequence of elements in an array of narrow character type (3.3151.1), where the size and alignment of the sequence match those of some object pointer type.

1

A pointer value is a *safely-derived pointer* to a dynamic object only if it has an object pointer type and it is one of the following:

— the value returned by a call to the C++ standard library implementation of `::operator new(std::size_t)`;[39]

38) Some implementations might define that copying an invalid pointer value causes a system-generated runtime fault.

39) This section does not impose restrictions on indirection through pointers to memory not allocated by `::operator new`. This maintains the ability of many C++ implementations to use binary libraries and components written in other languages. In particular, this applies to C binaries, because indirection through pointers to memory allocated by `std::malloc` is not restricted.

— the result of taking the address of an object (or one of its subobjects) designated by an lvalue resulting from indirection through a safely-derived pointer value;

— the result of well-defined pointer arithmetic (5.2882) using a safely-derived pointer value;

— the result of a well-defined pointer conversion (4.579, 5.2765) of a safely-derived pointer value;

— the result of a `reinterpret_cast` of a safely-derived pointer value;

— the result of a `reinterpret_cast` of an integer representation of a safely-derived pointer value;

— the value of an object whose value was copied from a traceable pointer object, where at the time of the copy the source object contained a copy of a safely-derived pointer value.

1

An integer value is an *integer representation of a safely-derived pointer* only if its type is at least as large as `std::intptr_t` and it is one of the following:

— the result of a `reinterpret_cast` of a safely-derived pointer value;

— the result of a valid conversion of an integer representation of a safely-derived pointer value;

— the value of an object whose value was copied from a traceable pointer object, where at the time of the copy the source object contained an integer representation of a safely-derived pointer value;

— the result of an additive or bitwise operation, one of whose operands is an integer representation of a safely-derived pointer value `P`, if that result converted by `reinterpret_cast<void*>` would compare equal to a safely-derived pointer computable from `reinterpret_cast<void*>(P)`.

1

An implementation may have *relaxed pointer safety*, in which case the validity of a pointer value does not depend on whether it is a safely-derived pointer value. Alternatively, an implementation may have *strict pointer safety*, in which case a pointer value referring to an object with dynamic storage duration that is not a safely-derived pointer value is an invalid pointer value unless the referenced complete object has previously been declared reachable (20.85.1). [*Note:* the effect of using an invalid pointer value (including passing it to a deallocation function) is undefined, see 3.2428.0.1. This is true even if the unsafely-derived pointer value might compare equal to some safely-derived pointer value. — *end note*] It is implementation defined whether an implementation has relaxed or strict pointer safety.

3.2554.1 Duration of subobjects [basic.stc.inherit]

1

The storage duration of member subobjects, base class subobjects and array elements is that of their complete object (1.319).

3.2563 Object lifetime [basic.life]

1

The *lifetime* of an object is a runtime property of the object. An object is said to have non-trivial initialization if it is of a class or aggregate type and it or one of its members is initialized by a constructor other than a trivial default constructor. [*Note:* initialization by a trivial copy/move constructor is non-trivial initialization. — *end note*] The lifetime of an object of type `T` begins when:

— storage with the proper alignment and size for type `T` is obtained, and

— if the object has non-trivial initialization, its initialization is complete.

The lifetime of an object of type `T` ends when:

— if `T` is a class type with a non-trivial destructor (12.807), the destructor call starts, or

— the storage which the object occupies is reused or released.

1

[*Note:* The lifetime of an array object starts as soon as storage with proper size and alignment is obtained, and its lifetime ends when the storage which the array occupies is reused or released. 12.1026.1 describes the lifetime of base and member subobjects. — *end note*]

1

The properties ascribed to objects throughout this International Standard apply for a given object only during its lifetime. [*Note:* In particular, before the lifetime of an object starts and after its lifetime ends there are significant restrictions on the use of the object, as described below, in 12.1026.1 and in 12.1483. Also, the behavior of an object under construction and destruction might not be the same as the behavior of an object whose lifetime has started and not ended. 12.1026.1 and 12.1483 describe the behavior of objects during the construction and destruction phases. — *end note*]

1

A program may end the lifetime of any object by reusing the storage which the object occupies or by explicitly calling the destructor for an object of a class type with a non-trivial destructor. For an object of a class type with a non-trivial destructor, the program is not required to call the destructor explicitly before the storage which the object occupies is reused or released; however, if there is no explicit call to the destructor or if a *delete-expression* (5.2455.1) is not used to release the storage, the destructor shall not be implicitly called and any program that depends on the side effects produced by the destructor has undefined behavior.

1

Before the lifetime of an object has started but after the storage which the object will occupy has been allocated[40] or, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, any pointer that refers to the storage location where the object will be or was located may be used but only in limited ways. For an object under construction or destruction, see 12.1483. Otherwise, such a pointer refers to allocated storage (3.2428.0.1), and using the pointer as if the pointer were of type `void*`, is well-defined. Indirection through such a pointer is permitted but the resulting lvalue may only be used in limited ways, as described below. The program has undefined behavior if:

— the object will be or was of a class type with a non-trivial destructor and the pointer is used as the operand of a *delete-expression*,

— the pointer is used to access a non-static data member or call a non-static member function of the object, or

— the pointer is implicitly converted (4.579) to a pointer to a virtual base class, or

— the pointer is used as the operand of a `static_cast` (5.1443.1), except when the conversion is to pointer to *cv* `void`, or to pointer to *cv* `void` and subsequently to pointer to either *cv* `char` or *cv* `unsigned char`, or

— the pointer is used as the operand of a `dynamic_cast` (5.1295.1). [*Example:*

```
#include <cstdlib>

struct B {
  virtual void f();
```

40) For example, before the construction of a global object of non-POD class type (12.1483).

```
  void mutate();
  virtual ~B();
};

struct D1 : B { void f(); };
struct D2 : B { void f(); };

void B::mutate() {
  new (this) D2;     // reuses storage — ends the lifetime of *this
  f();               // undefined behavior
  ... = this;        // OK, this points to valid memory
}

void g() {
  void* p = std::malloc(sizeof(D1) + sizeof(D2));
  B* pb = new (p) D1;
  pb->mutate();
  &pb;               // OK: pb points to valid memory
  void* q = pb;      // OK: pb points to valid memory
  pb->f();           // undefined behavior, lifetime of *pb has ended
}
```

— *end example*]

1

Similarly, before the lifetime of an object has started but after the storage which the object will occupy has been allocated or, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, any glvalue that refers to the original object may be used but only in limited ways. For an object under construction or destruction, see 12.1483. Otherwise, such a glvalue refers to allocated storage (3.2428.0.1), and using the properties of the glvalue that do not depend on its value is well-defined. The program has undefined behavior if:

— an lvalue-to-rvalue conversion (4.127) is applied to such a glvalue,

— the glvalue is used to access a non-static data member or call a non-static member function of the object, or

— the glvalue is bound to a reference to a virtual base class (8.2156.1), or

— the glvalue is used as the operand of a `dynamic_cast` (5.1295.1) or as the operand of `typeid`.

1

If, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, a new object is created at the storage location which the original object occupied, a pointer that pointed to the original object, a reference that referred to the original object, or the name of the original object will automatically refer to the new object and, once the lifetime of the new object has started, can be used to manipulate the new object, if:

— the storage for the new object exactly overlays the storage location which the original object occupied, and

— the new object is of the same type as the original object (ignoring the top-level cv-qualifiers), and

— the type of the original object is not const-qualified, and, if a class type, does not contain any non-static data member whose type is const-qualified or a reference type, and

— the original object was a most derived object (1.319) of type `T` and the new object is a most derived object of type `T` (that is, they are not base class subobjects). [*Example:*

```
struct C {
  int i;
  void f();
  const C& operator=( const C& );
};

const C& C::operator=( const C& other) {
  if ( this != &other ) {
    this->~C();                 // lifetime of *this ends
    new (this) C(other);        // new object of type C created
    f();                        // well-defined
  }
  return *this;
}

C c1;
C c2;
c1 = c2;                        // well-defined
c1.f();                         // well-defined; c1 refers to a new object of type C
```

— *end example*]

1

If a program ends the lifetime of an object of type `T` with static (3.2242.1), thread (3.2291.1), or automatic (3.2301.1) storage duration and if `T` has a non-trivial destructor,[41] the program must ensure that an object of the original type occupies that same storage location when the implicit destructor call takes place; otherwise the behavior of the program is undefined. This is true even if the block is exited with an exception. [*Example:*

```
class T { };
struct B {
   ~B();
};

void h() {
   B b;
   new (&b) T;
}                                // undefined behavior at block exit
```

— *end example*]

1

Creating a new object at the storage location that a `const` object with static, thread, or automatic storage duration occupies or, at the storage location that such a `const` object used to occupy before its lifetime ended results in undefined behavior. [*Example:*

```
struct B {
  B();
  ~B();
```

41) That is, an object for which a destructor will be called implicitly—upon exit from the block for an object with automatic storage duration, upon exit from the thread for an object with thread storage duration, or upon exit from the program for an object with static storage duration.

```
};

const B b;

void h() {
  b.~B();
  new (const_cast<B*>(&b)) const B;          // undefined behavior
}
```

— *end example*]

1 In this section, "before" and "after" refer to the "happens before" relation (1.859). [*Note:* Therefore, undefined behavior results if an object that is being constructed in one thread is referenced from another thread without adequate synchronization. — *end note*]

3.2955 Types [basic.types]

1

[*Note:* 3.2955 and the subclauses thereof impose requirements on implementations regarding the representation of types. There are two kinds of types: fundamental types and compound types. Types describe objects (1.319), references (8.97.1), or functions (8.360.1). — *end note*]

1

For any object (other than a base-class subobject) of trivially copyable type `T`, whether or not the object holds a valid value of type `T`, the underlying bytes (1.277) making up the object can be copied into an array of `char` or `unsigned char`.[42] If the content of the array of `char` or `unsigned char` is copied back into the object, the object shall subsequently hold its original value. [*Example:*

```
#define N sizeof(T)
char buf[N];
T obj;                          // obj initialized to its original value
std::memcpy(buf, &obj, N);      // between these two calls to std::memcpy,
                                // obj might be modified
std::memcpy(&obj, buf, N);      // at this point, each subobject of obj of scalar type
                                // holds its original value
```

— *end example*]

1

For any trivially copyable type `T`, if two pointers to `T` point to distinct `T` objects `obj1` and `obj2`, where neither `obj1` nor `obj2` is a base-class subobject, if the underlying bytes (1.277) making up `obj1` are copied into `obj2`,[43] `obj2` shall subsequently hold the same value as `obj1`. [*Example:*

```
T* t1p;
T* t2p;
    // provided that t2p points to an initialized object ...
std::memcpy(t1p, t2p, sizeof(T));
    // at this point, every subobject of trivially copyable type in *t1p contains
    // the same value as the corresponding subobject in *t2p
```

— *end example*]

1

42) By using, for example, the library functions (17.48.0.1) `std::memcpy` or `std::memmove`.
43) By using, for example, the library functions (17.48.0.1) `std::memcpy` or `std::memmove`.

The *object representation* of an object of type `T` is the sequence of N `unsigned char` objects taken up by the object of type `T`, where N equals `sizeof(T)`. The *value representation* of an object is the set of bits that hold the value of type `T`. For trivially copyable types, the value representation is a set of bits in the object representation that determines a *value*, which is one discrete element of an implementation-defined set of values.[44]

1

A class that has been declared but not defined, an enumeration type in certain contexts (7.1001), or an array of unknown size or of incomplete element type, is an *incompletely-defined object type*.[45] Incompletely-defined object types and the void types are *incomplete types* (3.3151.1). Objects shall not be defined to have an incomplete type.

1

A class type (such as "`class X`") might be incomplete at one point in a translation unit and complete later on; the type "`class X`" is the same type at both points. The declared type of an array object might be an array of incomplete class type and therefore incomplete; if the class type is completed later on in the translation unit, the array type becomes complete; the array type at those two points is the same type. The declared type of an array object might be an array of unknown size and therefore be incomplete at one point in a translation unit and complete later on; the array types at those two points ("array of unknown bound of `T`" and "array of `N T`") are different types. The type of a pointer to array of unknown size, or of a type defined by a `typedef` declaration to be an array of unknown size, cannot be completed. [*Example:*

```
class X;                        // X is an incomplete type
extern X* xp;                   // xp is a pointer to an incomplete type
extern int arr[];               // the type of arr is incomplete
typedef int UNKA[];             // UNKA is an incomplete type
UNKA* arrp;                     // arrp is a pointer to an incomplete type
UNKA** arrpp;

void foo() {
  xp++;                         // ill-formed: X is incomplete
  arrp++;                       // ill-formed: incomplete type
  arrpp++;                      // OK: sizeof UNKA* is known
}

struct X { int i; };            // now X is a complete type
int  arr[10];                   // now the type of arr is complete

X x;
void bar() {
  xp = &x;                      // OK; type is "pointer to X"
  arrp = &arr;                  // ill-formed: different types
  xp++;                         // OK: X is complete
  arrp++;                       // ill-formed: UNKA can't be completed
}
```

— *end example*]

1 [*Note:* The rules for declarations and expressions describe in which contexts incomplete types are prohibited. — *end note*]

2

44) The intent is that the memory model of C++ is compatible with that of ISO/IEC 9899 Programming Language C.
45) The size and layout of an instance of an incompletely-defined object type is unknown.

An *object type* is a (possibly cv-qualified) type that is not a function type, not a reference type, and not a void type.

1

Arithmetic types (3.3151.1), enumeration types, pointer types, pointer to member types (3.3568.1), `std::nullptr_t`, and cv-qualified versions of these types (3.3640.1) are collectively called *scalar types*. Scalar types, POD classes (Clause 9), arrays of such types and *cv-qualified* versions of these types (3.3640.1) are collectively called *POD types*. Cv-unqualified scalar types, trivially copyable class types (Clause 9), arrays of such types, and non-volatile const-qualified versions of these types (3.3640.1) are collectively called *trivially copyable types*. Scalar types, trivial class types (Clause 9), arrays of such types and cv-qualified versions of these types (3.3640.1) are collectively called *trivial types*. Scalar types, standard-layout class types (Clause 9), arrays of such types and cv-qualified versions of these types (3.3640.1) are collectively called *standard-layout types*.

1

A type is a *literal type* if it is:

— `void`; or

— a scalar type; or

— a reference type; or

— an array of literal type; or

— a class type (Clause 9) that has all of the following properties:

  — it has a trivial destructor,

  — it is an aggregate type (8.1817.1) or has at least one `constexpr` constructor or constructor template that is not a copy or move constructor, and

  — all of its non-static data members and base classes are of non-volatile literal types.

1

If two types `T1` and `T2` are the same type, then `T1` and `T2` are *layout-compatible* types. [*Note:* Layout-compatible enumerations are described in 7.1001. Layout-compatible standard-layout structs and standard-layout unions are described in 9.194. — *end note*]

3.3151.1 Fundamental types [basic.fundamental]

1

Objects declared as characters (`char`) shall be large enough to store any member of the implementation's basic character set. If a character from this set is stored in a character object, the integral value of that character object is equal to the value of the single character literal form of that character. It is implementation-defined whether a `char` object can hold negative values. Characters can be explicitly declared `unsigned` or `signed`. Plain `char`, `signed char`, and `unsigned char` are three distinct types, collectively called *narrow character types*. A `char`, a `signed char`, and an `unsigned char` occupy the same amount of storage and have the same alignment requirements (3.3924); that is, they have the same object representation. For narrow character types, all bits of the object representation participate in the value representation. For unsigned narrow character types, all possible bit patterns of the value representation represent numbers. These requirements do not hold for other types. In any particular implementation, a plain `char` object can take on either the same values as a `signed char` or an `unsigned char`; which one is implementation-defined. For each value i of type `unsigned char` in the range 0 to 255 inclusive, there exists a value j of type `char` such that the result of an integral conversion (4.397) from i to `char` is j, and the result of an integral conversion from j to `unsigned char` is i.

1

There are five *standard signed integer types* : "`signed char`", "`short int`", "`int`", "`long int`", and "`long long int`". In this list, each type provides at least as much storage as those preceding it in the list. There may also be implementation-defined *extended signed integer types*. The standard and extended signed integer types are collectively called *signed integer types*. Plain `int`s have the natural size suggested by the architecture of the execution environment[46]; the other signed integer types are provided to meet special needs.

1

For each of the standard signed integer types, there exists a corresponding (but different) *standard unsigned integer type*: "`unsigned char`", "`unsigned short int`", "`unsigned int`", "`unsigned long int`", and "`unsigned long long int`", each of which occupies the same amount of storage and has the same alignment requirements (3.3924) as the corresponding signed integer type[47]; that is, each signed integer type has the same object representation as its corresponding unsigned integer type. Likewise, for each of the extended signed integer types there exists a corresponding *extended unsigned integer type* with the same amount of storage and alignment requirements. The standard and extended unsigned integer types are collectively called *unsigned integer types*. The range of non-negative values of a *signed integer* type is a subrange of the corresponding *unsigned integer* type, and the value representation of each corresponding signed/unsigned type shall be the same. The standard signed integer types and standard unsigned integer types are collectively called the *standard integer types*, and the extended signed integer types and extended unsigned integer types are collectively called the *extended integer types*. The signed and unsigned integer types shall satisfy the constraints given in the C standard, section 5.2.4.2.1.

1

Unsigned integers shall obey the laws of arithmetic modulo 2^n where n is the number of bits in the value representation of that particular size of integer.[48]

1

Type `wchar_t` is a distinct type whose values can represent distinct codes for all members of the largest extended character set specified among the supported locales (22.3.1). Type `wchar_t` shall have the same size, signedness, and alignment requirements (3.3924) as one of the other integral types, called its *underlying type*. Types `char16_t` and `char32_t` denote distinct types with the same size, signedness, and alignment as `uint_least16_t` and `uint_least32_t`, respectively, in `<cstdint>`, called the underlying types.

1

Values of type `bool` are either `true` or `false`.[49] [*Note:* There are no `signed`, `unsigned`, `short`, or `long bool` types or values. — *end note*] Values of type `bool` participate in integral promotions (4.262).

1

Types `bool`, `char`, `char16_t`, `char32_t`, `wchar_t`, and the signed and unsigned integer types are collectively called *integral* types.[50] A synonym for integral type is *integer type*. The representations of integral types shall define values by use of a pure binary numeration system.[51] [*Example:* this International Standard

46) that is, large enough to contain any value in the range of `INT_MIN` and `INT_MAX`, as defined in the header `<climits>`.

47) See 7.678.0.1 regarding the correspondence between types and the sequences of *type-specifier*s that designate them.

48) This implies that unsigned arithmetic does not overflow because a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting unsigned integer type.

49) Using a `bool` value in ways described by this International Standard as "undefined," such as by examining the value of an uninitialized automatic object, might cause it to behave as if it is neither `true` nor `false`.

50) Therefore, enumerations (7.1001) are not integral; however, enumerations can be promoted to integral types as specified in 4.262.

51) A positional representation for integers that uses the binary digits 0 and 1, in which the values represented by successive bits are additive, begin with 1, and are multiplied by successive integral power of 2, except perhaps for the bit with the highest

permits 2's complement, 1's complement and signed magnitude representations for integral types. — *end example*]

1

There are three *floating point* types: `float`, `double`, and `long double`. The type `double` provides at least as much precision as `float`, and the type `long double` provides at least as much precision as `double`. The set of values of the type `float` is a subset of the set of values of the type `double`; the set of values of the type `double` is a subset of the set of values of the type `long double`. The value representation of floating-point types is implementation-defined. *Integral* and *floating* types are collectively called *arithmetic* types. Specializations of the standard template `std::numeric_limits` (18.41) shall specify the maximum and minimum values of each arithmetic type for an implementation.

1

The `void` type has an empty set of values. The `void` type is an incomplete type that cannot be completed. It is used as the return type for functions that do not return a value. Any expression can be explicitly converted to type *cv* `void` (5.2765). An expression of type `void` shall be used only as an expression statement (6.9), as an operand of a comma expression (5.3602), as a second or third operand of `?:` (5.3337), as the operand of `typeid`, `noexcept`, or `decltype`, as the expression in a return statement (6.270.2) for a function with the return type `void`, or as the operand of an explicit conversion to type *cv* `void`.

1 A value of type `std::nullptr_t` is a null pointer constant (4.579). Such values participate in the pointer and the pointer to member conversions (4.579, 4.710). `sizeof(std::nullptr_t)` shall be equal to `sizeof(void*)`.

2

[*Note:* Even if the implementation defines two or more basic types to have the same value representation, they are nevertheless different types. — *end note*]

3.3568.1 Compound types [basic.compound]

1

Compound types can be constructed in the following ways:

— *arrays* of objects of a given type, 8.251.1;

— *functions*, which have parameters of given types and return `void` or references or objects of a given type, 8.360.1;

— *pointers* to `void` or objects or functions (including static members of classes) of a given type, 8.94.1;

— *references* to objects or functions of a given type, 8.97.1. There are two types of references:

  — *lvalue reference*

  — *rvalue reference*

— *classes* containing a sequence of objects of various types (Clause 9), a set of types, enumerations and functions for manipulating these objects (9.401), and a set of restrictions on the access to these entities (Clause 11);

— *unions*, which are classes capable of containing objects of different types at different times, 9.815;

— *enumerations*, which comprise a set of named constant values. Each distinct enumeration constitutes a different *enumerated type*, 7.1001;

position. (Adapted from the *American National Dictionary for Information Processing Systems*.)

— *pointers to non-static* [52] *class members*, which identify members of a given type within objects of a given class, 8.236.1.

1

These methods of constructing types can be applied recursively; restrictions are mentioned in 8.94.1, 8.251.1, 8.360.1, and 8.97.1. Constructing a type such that the number of bytes in its object representation exceeds the maximum value representable in the type `std::size_t` (18.2) is ill-formed.

1

The type of a pointer to `void` or a pointer to an object type is called an *object pointer type*. [*Note:* A pointer to `void` does not have a pointer-to-object type, however, because `void` is not an object type. — *end note*] The type of a pointer that can designate a function is called a *function pointer type*. A pointer to objects of type `T` is referred to as a "pointer to `T`." [*Example:* a pointer to an object of type `int` is referred to as "pointer to `int` " and a pointer to an object of class `X` is called a "pointer to `X`." — *end example*] Except for pointers to static members, text referring to "pointers" does not apply to pointers to members. Pointers to incomplete types are allowed although there are restrictions on what can be done with them (3.3924). A valid value of an object pointer type represents either the address of a byte in memory (1.277) or a null pointer (4.579). If an object of type `T` is located at an address `A`, a pointer of type *cv* `T*` whose value is the address `A` is said to *point to* that object, regardless of how the value was obtained. [*Note:* For instance, the address one past the end of an array (5.2882) would be considered to point to an unrelated object of the array's element type that might be located at that address. There are further restrictions on pointers to objects with dynamic storage duration; see 3.2532.0.1. — *end note*] The value representation of pointer types is implementation-defined. Pointers to cv-qualified and cv-unqualified versions (3.3640.1) of layout-compatible types shall have the same value representation and alignment requirements (3.3924). [*Note:* Pointers to over-aligned types (3.3924) have no special representation, but their range of valid values is restricted by the extended alignment requirement. This International Standard specifies only two ways of obtaining such a pointer: taking the address of a valid object with an over-aligned type, and using one of the runtime pointer alignment functions. An implementation may provide other means of obtaining a valid pointer value for an over-aligned type. — *end note*]

1

A pointer to *cv*-qualified (3.3640.1) or *cv*-unqualified `void` can be used to point to objects of unknown type. Such a pointer shall be able to hold any object pointer. An object of type *cv* `void*` shall have the same representation and alignment requirements as *cv* `char*`.

3.3640.1 CV-qualifiers [basic.type.qualifier]

1

A type mentioned in 3.3151.1 and 3.3568.1 is a *cv-unqualified type*. Each type which is a cv-unqualified complete or incomplete object type or is `void` (3.2955) has three corresponding cv-qualified versions of its type: a *const-qualified* version, a *volatile-qualified* version, and a *const-volatile-qualified* version. The term *object type* (1.319) includes the cv-qualifiers specified in the *decl-specifier-seq* (7.40), *declarator* (Clause 8), *type-id* (8.32), or *new-type-id* (5.2221.1) when the object is created.

— A *const object* is an object of type `const T` or a non-mutable subobject of such an object.

— A *volatile object* is an object of type `volatile T`, a subobject of such an object, or a mutable subobject of a const volatile object.

— A *const volatile object* is an object of type `const volatile T`, a non-mutable subobject of such an object, a const subobject of a volatile object, or a non-mutable volatile subobject of a const object.

52) Static class members are objects or functions, and pointers to them are ordinary pointers to objects or functions.

The cv-qualified or cv-unqualified versions of a type are distinct types; however, they shall have the same representation and alignment requirements (3.3924).[53]

1

A compound type (3.3568.1) is not cv-qualified by the cv-qualifiers (if any) of the types from which it is compounded. Any cv-qualifiers applied to an array type affect the array element type, not the array type (8.251.1).

1 See 8.360.1 and 9.508.1 regarding function types that have *cv-qualifier*s.

2

There is a partial ordering on cv-qualifiers, so that a type can be said to be *more cv-qualified* than another. Table 9 shows the relations that constitute this ordering.

Table 9 — Relations on `const` and `volatile`

no cv-qualifier	<	`const`
no cv-qualifier	<	`volatile`
no cv-qualifier	<	`const volatile`
`const`	<	`const volatile`
`volatile`	<	`const volatile`

1

In this International Standard, the notation *cv* (or *cv1*, *cv2*, etc.), used in the description of types, represents an arbitrary set of cv-qualifiers, i.e., one of {`const`}, {`volatile`}, {`const`, `volatile`}, or the empty set. Cv-qualifiers applied to an array type attach to the underlying element type, so the notation "*cv* `T`," where `T` is an array type, refers to an array whose elements are so-qualified. An array type whose elements are cv-qualified is also considered to have the same cv-qualifications as its elements. [*Example:*

```
typedef char CA[5];
typedef const char CC;
CC arr1[5] = { 0 };
const CA arr2 = { 0 };
```

The type of both `arr1` and `arr2` is "array of 5 `const char`," and the array type is considered to be `const`-qualified. — *end example*]

3.3673 Lvalues and rvalues [basic.lval]

1

Expressions are categorized according to the taxonomy in Figure 1.

— An *lvalue* (so called, historically, because lvalues could appear on the left-hand side of an assignment expression) designates a function or an object. [*Example:* If `E` is an expression of pointer type, then `*E` is an lvalue expression referring to the object or function to which `E` points. As another example, the result of calling a function whose return type is an lvalue reference is an lvalue. — *end example*]

— An *xvalue* (an "eXpiring" value) also refers to an object, usually near the end of its lifetime (so that its resources may be moved, for example). An xvalue is the result of certain kinds of expressions involving rvalue references (8.97.1). [*Example:* The result of calling a function whose return type is an rvalue reference to an object type is an xvalue (5.955.1). — *end example*]

— A *glvalue* ("generalized" lvalue) is an lvalue or an xvalue.

53) The same representation and alignment requirements are meant to imply interchangeability as arguments to functions, return values from functions, and non-static data members of unions.

Figure 1 — Expression category taxonomy

— An *rvalue* (so called, historically, because rvalues could appear on the right-hand side of an assignment expression) is an xvalue, a temporary object (12.306) or subobject thereof, or a value that is not associated with an object.

— A *prvalue* ("pure" rvalue) is an rvalue that is not an xvalue. [*Example:* The result of calling a function whose return type is not a reference is a prvalue. The value of a literal such as `12`, `7.3e5`, or `true` is also a prvalue. — *end example*]

Every expression belongs to exactly one of the fundamental classifications in this taxonomy: lvalue, xvalue, or prvalue. This property of an expression is called its *value category*. [*Note:* The discussion of each built-in operator in Clause 5 indicates the category of the value it yields and the value categories of the operands it expects. For example, the built-in assignment operators expect that the left operand is an lvalue and that the right operand is a prvalue and yield an lvalue as the result. User-defined operators are functions, and the categories of values they expect and yield are determined by their parameter and return types. — *end note*]

1

Whenever a glvalue appears in a context where a prvalue is expected, the glvalue is converted to a prvalue; see 4.127, 4.180, and 4.220. [*Note:* An attempt to bind an rvalue reference to an lvalue is not such a context; see 8.2156.1. — *end note*]

1

The discussion of reference initialization in 8.2156.1 and of temporaries in 12.306 indicates the behavior of lvalues and rvalues in other significant contexts.

1

Unless otherwise indicated (5.955.1), prvalues shall always have complete types or the `void` type; in addition to these types, glvalues can also have incomplete types. [*Note:* class and array prvalues can have cv-qualified types; other prvalues always have cv-unqualified types. See Clause 5. — *end note*]

1

An lvalue for an object is necessary in order to modify the object except that an rvalue of class type can also be used to modify its referent under certain circumstances. [*Example:* a member function called for an object (9.401) can modify the object. — *end example*]

1

Functions cannot be modified, but pointers to functions can be modifiable.

1

A pointer to an incomplete type can be modifiable. At some point in the program when the pointed to type is complete, the object at which the pointer points can also be modified.

1

The referent of a `const`-qualified expression shall not be modified (through that expression), except that if it is of class type and has a `mutable` component, that component can be modified (7.567.0.1).

1

If an expression can be used to modify the object to which it refers, the expression is called *modifiable*. A program that attempts to modify an object through a nonmodifiable lvalue or rvalue expression is ill-formed.

1

If a program attempts to access the stored value of an object through a glvalue of other than one of the following types the behavior is undefined:[54]

— the dynamic type of the object,

— a cv-qualified version of the dynamic type of the object,

— a type similar (as defined in 4.246) to the dynamic type of the object,

— a type that is the signed or unsigned type corresponding to the dynamic type of the object,

— a type that is the signed or unsigned type corresponding to a cv-qualified version of the dynamic type of the object,

— an aggregate or union type that includes one of the aforementioned types among its elements or non-static data members (including, recursively, an element or non-static data member of a subaggregate or contained union),

— a type that is a (possibly cv-qualified) base class type of the dynamic type of the object,

— a `char` or `unsigned char` type.

3.3924 Alignment [basic.align]

1

Object types have *alignment requirements* (3.3151.1, 3.3568.1) which place restrictions on the addresses at which an object of that type may be allocated. An *alignment* is an implementation-defined integer value representing the number of bytes between successive addresses at which a given object can be allocated. An object type imposes an alignment requirement on every object of that type; stricter alignment can be requested using the alignment specifier (7.1556.1).

1

A *fundamental alignment* is represented by an alignment less than or equal to the greatest alignment supported by the implementation in all contexts, which is equal to `alignof(std::max_align_t)` (18.2). The alignment required for a type might be different when it is used as the type of a complete object and when it is used as the type of a subobject. [*Example:*

```
struct B { long double d; };
struct D : virtual B { char c; }
```

When `D` is the type of a complete object, it will have a subobject of type `B`, so it must be aligned appropriately for a `long double`. If `D` appears as a subobject of another object that also has `B` as a virtual base class, the `B` subobject might be part of a different subobject, reducing the alignment requirements on the `D` subobject. — *end example*] The result of the `alignof` operator reflects the alignment requirement of the type in the complete-object case.

54) The intent of this list is to specify those circumstances in which an object may or may not be aliased.

1

An *extended alignment* is represented by an alignment greater than `alignof(std::max_align_t)`. It is implementation-defined whether any extended alignments are supported and the contexts in which they are supported (7.1556.1). A type having an extended alignment requirement is an *over-aligned type*. [*Note:* every over-aligned type is or contains a class type to which extended alignment applies (possibly through a non-static data member). — *end note*]

1

Alignments are represented as values of the type `std::size_t`. Valid alignments include only those values returned by an `alignof` expression for the fundamental types plus an additional implementation-defined set of values, which may be empty. Every alignment value shall be a non-negative integral power of two.

1

Alignments have an order from *weaker* to *stronger* or *stricter* alignments. Stricter alignments have larger alignment values. An address that satisfies an alignment requirement also satisfies any weaker valid alignment requirement.

1

The alignment requirement of a complete type can be queried using an `alignof` expression (5.2747.1). Furthermore, the narrow character types (3.3151.1) shall have the weakest alignment requirement. [*Note:* This enables the narrow character types to be used as the underlying type for an aligned memory area (7.1556.1). — *end note*]

1

Comparing alignments is meaningful and provides the obvious results:

— Two alignments are equal when their numeric values are equal.

— Two alignments are different when their numeric values are not equal.

— When an alignment is larger than another it represents a stricter alignment.

1 [*Note:* The runtime pointer alignment function (20.85.2) can be used to obtain an aligned pointer within a buffer; the aligned-storage templates in the library (20.570.1.6) can be used to obtain aligned storage. — *end note*]

2

If a request for a specific extended alignment in a specific context is not supported by an implementation, the program is ill-formed. Additionally, a request for runtime allocation of dynamic storage for which the requested alignment cannot be honored shall be treated as an allocation failure.

4 Standard conversions [conv]

1

Standard conversions are implicit conversions with built-in meaning. Clause 4 enumerates the full set of such conversions. A *standard conversion sequence* is a sequence of standard conversions in the following order:

— Zero or one conversion from the following set: lvalue-to-rvalue conversion, array-to-pointer conversion, and function-to-pointer conversion.

— Zero or one conversion from the following set: integral promotions, floating point promotion, integral conversions, floating point conversions, floating-integral conversions, pointer conversions, pointer to member conversions, and boolean conversions.

— Zero or one qualification conversion.

[*Note:* A standard conversion sequence can be empty, i.e., it can consist of no conversions. — *end note*] A standard conversion sequence will be applied to an expression if necessary to convert it to a required destination type.

1

[*Note:* expressions with a given type will be implicitly converted to other types in several contexts:

— When used as operands of operators. The operator's requirements for its operands dictate the destination type (Clause 5).

— When used in the condition of an `if` statement or iteration statement (6.23, 6.137). The destination type is `bool`.

— When used in the expression of a `switch` statement. The destination type is integral (6.23).

— When used as the source expression for an initialization (which includes use as an argument in a function call and use as the expression in a `return` statement). The type of the entity being initialized is (generally) the destination type. See 8.821, 8.2156.1.

— *end note*]

1

An expression `e` can be *implicitly converted* to a type `T` if and only if the declaration `T t=e;` is well-formed, for some invented temporary variable `t` (8.821).

1

Certain language constructs require that an expression be converted to a Boolean value. An expression `e` appearing in such a context is said to be *contextually converted to* `bool` and is well-formed if and only if the declaration `bool t(e);` is well-formed, for some invented temporary variable `t` (8.821).

1

Certain language constructs require conversion to a value having one of a specified set of types appropriate to the construct. An expression `e` of class type `E` appearing in such a context is said to be *contextually implicitly converted to* a specified type `T` and is well-formed if and only if `e` can be implicitly converted to a type `T` that is determined as follows: `E` is searched for conversion functions whose return type is *cv* `T` or reference to *cv* `T` such that `T` is allowed by the context. There shall be exactly one such `T`.

1

The effect of any implicit conversion is the same as performing the corresponding declaration and initialization and then using the temporary variable as the result of the conversion. The result is an lvalue if `T` is an lvalue reference type or an rvalue reference to function type (8.97.1), an xvalue if `T` is an rvalue reference to object type, and a prvalue otherwise. The expression `e` is used as a glvalue if and only if the initialization uses it as a glvalue.

1

[*Note:* For class types, user-defined conversions are considered as well; see 12.659. In general, an implicit conversion sequence (13.779.0.1) consists of a standard conversion sequence followed by a user-defined conversion followed by another standard conversion sequence. — *end note*]

1

[*Note:* There are some contexts where certain conversions are suppressed. For example, the lvalue-to-rvalue conversion is not done on the operand of the unary `&` operator. Specific exceptions are given in the descriptions of those operators and contexts. — *end note*]

4.127 Lvalue-to-rvalue conversion [conv.lval]

1

A glvalue (3.3673) of a non-function, non-array type `T` can be converted to a prvalue.[55] If `T` is an incomplete type, a program that necessitates this conversion is ill-formed. If `T` is a non-class type, the type of the prvalue is the cv-unqualified version of `T`. Otherwise, the type of the prvalue is `T`.[56]

1

When an lvalue-to-rvalue conversion is applied to an expression `e`, and either

— `e` is not potentially evaluated, or

— the evaluation of `e` results in the evaluation of a member `ex` of the set of potential results of `e`, and `ex` names a variable `x` that is not odr-used by `ex` (3.91),

the value contained in the referenced object is not accessed. [*Example:*

```
struct S { int n; };
auto f() {
  S x { 1 };
  constexpr S y { 2 };
  return [&](bool b) { return (b ? y : x).n; };
}
auto g = f();
int m = g(false); // undefined behavior due to access of x.n outside its lifetime
int n = g(true);  // OK, does not access y.n
```

— *end example*] In all other cases, the result of the conversion is determined according to the following rules:

— If `T` is (possibly cv-qualified) `std::nullptr_t`, the result is a null pointer constant (4.579).

— Otherwise, if `T` has a class type, the conversion copy-initializes a temporary of type `T` from the glvalue and the result of the conversion is a prvalue for the temporary.

55) For historical reasons, this conversion is called the "lvalue-to-rvalue" conversion, even though that name does not accurately reflect the taxonomy of expressions described in 3.3673.

56) In C++ class prvalues can have cv-qualified types (because they are objects). This differs from ISO C, in which non-lvalues never have cv-qualified types.

— Otherwise, if the object to which the glvalue refers contains an invalid pointer value (3.2428.0.1, 3.2532.0.1), the behavior is implementation-defined.

— Otherwise, the value contained in the object indicated by the glvalue is the prvalue result.

1 [*Note:* See also 3.3673. — *end note*]

4.180 Array-to-pointer conversion [conv.array]

1

An lvalue or rvalue of type "array of `N T`" or "array of unknown bound of `T`" can be converted to a prvalue of type "pointer to `T`". The result is a pointer to the first element of the array.

4.220 Function-to-pointer conversion [conv.func]

1

An lvalue of function type `T` can be converted to a prvalue of type "pointer to `T`." The result is a pointer to the function.[57]

1 [*Note:* See 13.1084 for additional rules for the case where the function is overloaded. — *end note*]

4.246 Qualification conversions [conv.qual]

1

A prvalue of type "pointer to *cv1* `T`" can be converted to a prvalue of type "pointer to *cv2* `T`" if "*cv2* `T`" is more cv-qualified than "*cv1* `T`".

1 A prvalue of type "pointer to member of `X` of type *cv1* `T`" can be converted to a prvalue of type "pointer to member of `X` of type *cv2* `T`" if "*cv2* `T`" is more cv-qualified than "*cv1* `T`".

2

[*Note:* Function types (including those used in pointer to member function types) are never cv-qualified (8.360.1). — *end note*]

1

A conversion can add cv-qualifiers at levels other than the first in multi-level pointers, subject to the following rules:[58]

> Two pointer types T1 and T2 are *similar* if there exists a type T and integer $n > 0$ such that:
>
> *T1* is $cv_{1,0}$ pointer to $cv_{1,1}$ pointer to $\cdots$ $cv_{1,n-1}$ pointer to $cv_{1,n}$ T
>
> and
>
> *T2* is $cv_{2,0}$ pointer to $cv_{2,1}$ pointer to $\cdots$ $cv_{2,n-1}$ pointer to $cv_{2,n}$ T
>
> where each $cv_{i,j}$ is `const`, `volatile`, `const volatile`, or nothing. The n-tuple of cv-qualifiers after the first in a pointer type, e.g., $cv_{1,1}$, $cv_{1,2}$, $\cdots$, $cv_{1,n}$ in the pointer type *T1*, is called the *cv-qualification signature* of the pointer type. An expression of type *T1* can be converted to type *T2* if and only if the following conditions are satisfied:
>
> — the pointer types are similar.
>
> — for every $j > 0$, if `const` is in $cv_{1,j}$ then `const` is in $cv_{2,j}$, and similarly for `volatile`.
>
> — if the $cv_{1,j}$ and $cv_{2,j}$ are different, then `const` is in every $cv_{2,k}$ for $0 < k < j$.

57) This conversion never applies to non-static member functions because an lvalue that refers to a non-static member function cannot be obtained.

58) These rules ensure that const-safety is preserved by the conversion.

[*Note:* if a program could assign a pointer of type `T**` to a pointer of type `const T**` (that is, if line `#1` below were allowed), a program could inadvertently modify a `const` object (as it is done on line `#2`). For example,

```
int main() {
  const char c = 'c';
  char* pc;
  const char** pcc = &pc;        // #1: not allowed
  *pcc = &c;
  *pc = 'C';                     // #2: modifies a const object
}
```

— *end note*]

1 A *multi-level* pointer to member type, or a *multi-level mixed* pointer and pointer to member type has the form:

$$cv_0 P_0 \text{ to } cv_1 P_1 \text{ to } \cdots\ cv_{n-1} P_{n-1} \text{ to } cv_n\ T$$

where P_i is either a pointer or pointer to member and where T is not a pointer type or pointer to member type.

2 Two multi-level pointer to member types or two multi-level mixed pointer and pointer to member types T1 and T2 are *similar* if there exists a type T and integer $n > 0$ such that:

$$T1 \text{ is } cv_{1,0} P_0 \text{ to } cv_{1,1} P_1 \text{ to } \cdots\ cv_{1,n-1} P_{n-1} \text{ to } cv_{1,n}\ T$$

and

$$T2 \text{ is } cv_{2,0} P_0 \text{ to } cv_{2,1} P_1 \text{ to } \cdots\ cv_{2,n-1} P_{n-1} \text{ to } cv_{2,n}\ T$$

3 For similar multi-level pointer to member types and similar multi-level mixed pointer and pointer to member types, the rules for adding cv-qualifiers are the same as those used for similar pointer types.

4.262 Integral promotions [conv.prom]

1

A prvalue of an integer type other than `bool`, `char16_t`, `char32_t`, or `wchar_t` whose integer conversion rank (4.807) is less than the rank of `int` can be converted to a prvalue of type `int` if `int` can represent all the values of the source type; otherwise, the source prvalue can be converted to a prvalue of type `unsigned int`.

1

A prvalue of type `char16_t`, `char32_t`, or `wchar_t` (3.3151.1) can be converted to a prvalue of the first of the following types that can represent all the values of its underlying type: `int`, `unsigned int`, `long int`, `unsigned long int`, `long long int`, or `unsigned long long int`. If none of the types in that list can represent all the values of its underlying type, a prvalue of type `char16_t`, `char32_t`, or `wchar_t` can be converted to a prvalue of its underlying type.

1

A prvalue of an unscoped enumeration type whose underlying type is not fixed (7.1001) can be converted to a prvalue of the first of the following types that can represent all the values of the enumeration (i.e., the values in the range b_{min} to b_{max} as described in 7.1001): `int`, `unsigned int`, `long int`, `unsigned long int`, `long long int`, or `unsigned long long int`. If none of the types in that list can represent all the values of the enumeration, a prvalue of an unscoped enumeration type can be converted to a prvalue of the extended integer type with lowest integer conversion rank (4.807) greater than the rank of `long long` in which all the values of the enumeration can be represented. If there are two such extended types, the signed one is chosen.

1

A prvalue of an unscoped enumeration type whose underlying type is fixed (7.1001) can be converted to a prvalue of its underlying type. Moreover, if integral promotion can be applied to its underlying type, a prvalue of an unscoped enumeration type whose underlying type is fixed can also be converted to a prvalue of the promoted underlying type.

1

A prvalue for an integral bit-field (9.921) can be converted to a prvalue of type `int` if `int` can represent all the values of the bit-field; otherwise, it can be converted to `unsigned int` if `unsigned int` can represent all the values of the bit-field. If the bit-field is larger yet, no integral promotion applies to it. If the bit-field has an enumerated type, it is treated as any other value of that type for promotion purposes.

1

A prvalue of type `bool` can be converted to a prvalue of type `int`, with `false` becoming zero and `true` becoming one.

1 These conversions are called *integral promotions*.

4.389 Floating point promotion [conv.fpprom]

1

A prvalue of type `float` can be converted to a prvalue of type `double`. The value is unchanged.

1 This conversion is called *floating point promotion*.

4.397 Integral conversions [conv.integral]

1

A prvalue of an integer type can be converted to a prvalue of another integer type. A prvalue of an unscoped enumeration type can be converted to a prvalue of an integer type.

1

If the destination type is unsigned, the resulting value is the least unsigned integer congruent to the source integer (modulo 2^n where n is the number of bits used to represent the unsigned type). [*Note:* In a two's complement representation, this conversion is conceptual and there is no change in the bit pattern (if there is no truncation). — *end note*]

1

If the destination type is signed, the value is unchanged if it can be represented in the destination type (and bit-field width); otherwise, the value is implementation-defined.

1

If the destination type is `bool`, see 4.732. If the source type is `bool`, the value `false` is converted to zero and the value `true` is converted to one.

1

The conversions allowed as integral promotions are excluded from the set of integral conversions.

4.516 Floating point conversions [conv.double]

1

A prvalue of floating point type can be converted to a prvalue of another floating point type. If the source value can be exactly represented in the destination type, the result of the conversion is that exact representation. If the source value is between two adjacent destination values, the result of the conversion is an implementation-defined choice of either of those values. Otherwise, the behavior is undefined.

1

The conversions allowed as floating point promotions are excluded from the set of floating point conversions.

4.532 Floating-integral conversions [conv.fpint]

1

A prvalue of a floating point type can be converted to a prvalue of an integer type. The conversion truncates; that is, the fractional part is discarded. The behavior is undefined if the truncated value cannot be represented in the destination type. [*Note:* If the destination type is `bool`, see 4.732. — *end note*]

1

A prvalue of an integer type or of an unscoped enumeration type can be converted to a prvalue of a floating point type. The result is exact if possible. If the value being converted is in the range of values that can be represented but the value cannot be represented exactly, it is an implementation-defined choice of either the next lower or higher representable value. [*Note:* Loss of precision occurs if the integral value cannot be represented exactly as a value of the floating type. — *end note*] If the value being converted is outside the range of values that can be represented, the behavior is undefined. If the source type is `bool`, the value `false` is converted to zero and the value `true` is converted to one.

4.579 Pointer conversions [conv.ptr]

1

A *null pointer constant* is an integer literal (2.300.1) with value zero or a prvalue of type `std::nullptr_t`. A null pointer constant can be converted to a pointer type; the result is the *null pointer value* of that type and is distinguishable from every other value of object pointer or function pointer type. Such a conversion is called a *null pointer conversion*. Two null pointer values of the same type shall compare equal. The conversion of a null pointer constant to a pointer to cv-qualified type is a single conversion, and not the sequence of a pointer conversion followed by a qualification conversion (4.246). A null pointer constant of integral type can be converted to a prvalue of type `std::nullptr_t`. [*Note:* The resulting prvalue is not a null pointer value. — *end note*]

1

A prvalue of type "pointer to *cv* `T`," where `T` is an object type, can be converted to a prvalue of type "pointer to *cv* `void`". The result of converting a non-null pointer value of a pointer to object type to a "pointer to *cv* `void`" represents the address of the same byte in memory as the original pointer value. The null pointer value is converted to the null pointer value of the destination type.

1

A prvalue of type "pointer to *cv* `D`", where `D` is a class type, can be converted to a prvalue of type "pointer to *cv* `B`", where `B` is a base class (Clause 10) of `D`. If `B` is an inaccessible (Clause 11) or ambiguous (10.52) base class of `D`, a program that necessitates this conversion is ill-formed. The result of the conversion is a pointer to the base class subobject of the derived class object. The null pointer value is converted to the null pointer value of the destination type.

4.710 Pointer to member conversions [conv.mem]

1

A null pointer constant (4.579) can be converted to a pointer to member type; the result is the *null member pointer value* of that type and is distinguishable from any pointer to member not created from a null pointer constant. Such a conversion is called a *null member pointer conversion*. Two null member pointer values of the same type shall compare equal. The conversion of a null pointer constant to a pointer to member of cv-qualified type is a single conversion, and not the sequence of a pointer to member conversion followed by a qualification conversion (4.246).

1

A prvalue of type "pointer to member of `B` of type *cv* `T`", where `B` is a class type, can be converted to a prvalue of type "pointer to member of `D` of type *cv* `T`", where `D` is a derived class (Clause 10) of `B`. If `B` is an inaccessible (Clause 11), ambiguous (10.52), or virtual (10.31) base class of `D`, or a base class of a virtual base class of `D`, a program that necessitates this conversion is ill-formed. The result of the conversion refers to the same member as the pointer to member before the conversion took place, but it refers to the base class member as if it were a member of the derived class. The result refers to the member in `D`'s instance of `B`. Since the result has type "pointer to member of `D` of type *cv* `T`", indirection through it with a `D` object is valid. The result is the same as if indirecting through the pointer to member of `B` with the `B` subobject of `D`. The null member pointer value is converted to the null member pointer value of the destination type.[59]

4.732 Boolean conversions [conv.bool]

1

A prvalue of arithmetic, unscoped enumeration, pointer, or pointer to member type can be converted to a prvalue of type `bool`. A zero value, null pointer value, or null member pointer value is converted to `false`; any other value is converted to `true`. For direct-initialization (8.821), a prvalue of type `std::nullptr_t` can be converted to a prvalue of type `bool`; the resulting value is `false`.

4.807 Integer conversion rank [conv.rank]

1

Every integer type has an *integer conversion rank* defined as follows:

— No two signed integer types other than `char` and `signed char` (if `char` is signed) shall have the same rank, even if they have the same representation.

— The rank of a signed integer type shall be greater than the rank of any signed integer type with a smaller size.

— The rank of `long long int` shall be greater than the rank of `long int`, which shall be greater than the rank of `int`, which shall be greater than the rank of `short int`, which shall be greater than the rank of `signed char`.

— The rank of any unsigned integer type shall equal the rank of the corresponding signed integer type.

— The rank of any standard integer type shall be greater than the rank of any extended integer type with the same size.

— The rank of `char` shall equal the rank of `signed char` and `unsigned char`.

— The rank of `bool` shall be less than the rank of all other standard integer types.

— The ranks of `char16_t`, `char32_t`, and `wchar_t` shall equal the ranks of their underlying types (3.3151.1).

— The rank of any extended signed integer type relative to another extended signed integer type with the same size is implementation-defined, but still subject to the other rules for determining the integer conversion rank.

— For all integer types `T1`, `T2`, and `T3`, if `T1` has greater rank than `T2` and `T2` has greater rank than `T3`, then `T1` shall have greater rank than `T3`.

59) The rule for conversion of pointers to members (from pointer to member of base to pointer to member of derived) appears inverted compared to the rule for pointers to objects (from pointer to derived to pointer to base) (4.579, Clause 10). This inversion is necessary to ensure type safety. Note that a pointer to member is not an object pointer or a function pointer and the rules for conversions of such pointers do not apply to pointers to members. In particular, a pointer to member cannot be converted to a `void*`.

[*Note:* The integer conversion rank is used in the definition of the integral promotions (4.262) and the usual arithmetic conversions (Clause 5). — *end note*]

5 Expressions [expr]

1

[*Note:* Clause 5 defines the syntax, order of evaluation, and meaning of expressions.[60] An expression is a sequence of operators and operands that specifies a computation. An expression can result in a value and can cause side effects. — *end note*]

1

[*Note:* Operators can be overloaded, that is, given meaning when applied to expressions of class type (Clause 9) or enumeration type (7.1001). Uses of overloaded operators are transformed into function calls as described in 13.1122. Overloaded operators obey the rules for syntax specified in Clause 5, but the requirements of operand type, value category, and evaluation order are replaced by the rules for function call. Relations between operators, such as `++a` meaning `a+=1`, are not guaranteed for overloaded operators (13.1122), and are not guaranteed for operands of type `bool`. — *end note*]

1

Clause 5 defines the effects of operators when applied to types for which they have not been overloaded. Operator overloading shall not modify the rules for the *built-in operators*, that is, for operators applied to types for which they are defined by this Standard. However, these built-in operators participate in overload resolution, and as part of that process user-defined conversions will be considered where necessary to convert the operands to types appropriate for the built-in operator. If a built-in operator is selected, such conversions will be applied to the operands before the operation is considered further according to the rules in Clause 5; see 13.264.0.1, 13.1258.

1

If during the evaluation of an expression, the result is not mathematically defined or not in the range of representable values for its type, the behavior is undefined. [*Note:* most existing implementations of C++ ignore integer overflows. Treatment of division by zero, forming a remainder using a zero divisor, and all floating point exceptions vary among machines, and is usually adjustable by a library function. — *end note*]

1

If an expression initially has the type "reference to `T`" (8.97.1, 8.2156.1), the type is adjusted to `T` prior to any further analysis. The expression designates the object or function denoted by the reference, and the expression is an lvalue or an xvalue, depending on the expression.

1

If a prvalue initially has the type "*cv* `T`," where `T` is a cv-unqualified non-class, non-array type, the type of the expression is adjusted to `T` prior to any further analysis.

1

[*Note:* An expression is an xvalue if it is:

— the result of calling a function, whether implicitly or explicitly, whose return type is an rvalue reference to object type,

— a cast to an rvalue reference to object type,

60) The precedence of operators is not directly specified, but it can be derived from the syntax.

— a class member access expression designating a non-static data member of non-reference type in which the object expression is an xvalue, or

— a `.*` pointer-to-member expression in which the first operand is an xvalue and the second operand is a pointer to data member.

In general, the effect of this rule is that named rvalue references are treated as lvalues and unnamed rvalue references to objects are treated as xvalues; rvalue references to functions are treated as lvalues whether named or not. — *end note*]

[*Example:*

```
struct A {
  int m;
};
A&& operator+(A, A);
A&& f();

A a;
A&& ar = static_cast<A&&>(a);
```

The expressions `f()`, `f().m`, `static_cast<A&&>(a)`, and `a + a` are xvalues. The expression `ar` is an lvalue. — *end example*]

1

In some contexts, *unevaluated operands* appear (5.1393.1, 5.2075.1, 5.2750.1, 7.678.0.1). An unevaluated operand is not evaluated. An unevaluated operand is considered a full-expression. [*Note:* In an unevaluated operand, a non-static class member may be named (5.398) and naming of objects or functions does not, by itself, require that a definition be provided (3.91). — *end note*]

1

Whenever a glvalue expression appears as an operand of an operator that expects a prvalue for that operand, the lvalue-to-rvalue (4.127), array-to-pointer (4.180), or function-to-pointer (4.220) standard conversions are applied to convert the expression to a prvalue. [*Note:* because cv-qualifiers are removed from the type of an expression of non-class type when the expression is converted to a prvalue, an lvalue expression of type `const int` can, for example, be used where a prvalue expression of type `int` is required. — *end note*]

1

Many binary operators that expect operands of arithmetic or enumeration type cause conversions and yield result types in a similar way. The purpose is to yield a common type, which is also the type of the result. This pattern is called the *usual arithmetic conversions*, which are defined as follows:

— If either operand is of scoped enumeration type (7.1001), no conversions are performed; if the other operand does not have the same type, the expression is ill-formed.

— If either operand is of type `long double`, the other shall be converted to `long double`.

— Otherwise, if either operand is `double`, the other shall be converted to `double`.

— Otherwise, if either operand is `float`, the other shall be converted to `float`.

— Otherwise, the integral promotions (4.262) shall be performed on both operands.[61] Then the following rules shall be applied to the promoted operands:

61) As a consequence, operands of type `bool`, `char16_t`, `char32_t`, `wchar_t`, or an enumerated type are converted to some integral type.

— If both operands have the same type, no further conversion is needed.

— Otherwise, if both operands have signed integer types or both have unsigned integer types, the operand with the type of lesser integer conversion rank shall be converted to the type of the operand with greater rank.

— Otherwise, if the operand that has unsigned integer type has rank greater than or equal to the rank of the type of the other operand, the operand with signed integer type shall be converted to the type of the operand with unsigned integer type.

— Otherwise, if the type of the operand with signed integer type can represent all of the values of the type of the operand with unsigned integer type, the operand with unsigned integer type shall be converted to the type of the operand with signed integer type.

— Otherwise, both operands shall be converted to the unsigned integer type corresponding to the type of the operand with signed integer type.

1

In some contexts, an expression only appears for its side effects. Such an expression is called a *discarded-value expression*. The expression is evaluated and its value is discarded. The array-to-pointer (4.180) and function-to-pointer (4.220) standard conversions are not applied. The lvalue-to-rvalue conversion (4.127) is applied if and only if the expression is a glvalue of volatile-qualified type and it is one of the following:

— `(` *expression* `)`, where *expression* is one of these expressions,

— *id-expression* (5.398.1),

— subscripting (5.896.1),

— class member access (5.1194.1),

— indirection (5.1898.1),

— pointer-to-member operation (5.2832),

— conditional expression (5.3337) where both the second and the third operands are one of these expressions, or

— comma expression (5.3602) where the right operand is one of these expressions.

[*Note:* Using an overloaded operator causes a function call; the above covers only operators with built-in meaning. If the lvalue is of class type, it must have a volatile copy constructor to initialize the temporary that is the result of the lvalue-to-rvalue conversion. — *end note*]

1

The values of the floating operands and the results of floating expressions may be represented in greater precision and range than that required by the type; the types are not changed thereby.[62]

1

The *cv-combined type* of two types `T1` and `T2` is a type `T3` similar to `T1` whose cv-qualification signature (4.246) is:

— for every $j > 0$, $cv_{3,j}$ is the union of $cv_{1,j}$ and $cv_{2,j}$;

— if the resulting $cv_{3,j}$ is different from $cv_{1,j}$ or $cv_{2,j}$, then `const` is added to every $cv_{3,k}$ for $0 < k < j$.

62) The cast and assignment operators must still perform their specific conversions as described in 5.2765, 5.1443.1 and 5.3483.

[*Note:* Given similar types `T1` and `T2`, this construction ensures that both can be converted to `T3`. — *end note*] The *composite pointer type* of two operands `p1` and `p2` having types `T1` and `T2`, respectively, where at least one is a pointer or pointer to member type or `std::nullptr_t`, is:

— if both `p1` and `p2` are null pointer constants, `std::nullptr_t`;

— if either `p1` or `p2` is a null pointer constant, `T2` or `T1`, respectively;

— if `T1` or `T2` is "pointer to *cv1* `void`" and the other type is "pointer to *cv2* T", "pointer to *cv12* `void`", where *cv12* is the union of *cv1* and *cv2*;

— if `T1` is "pointer to *cv1* `C1`" and `T2` is "pointer to *cv2* `C2`", where `C1` is reference-related to `C2` or `C2` is reference-related to `C1` (8.2156.1), the cv-combined type of `T1` and `T2` or the cv-combined type of `T2` and `T1`, respectively;

— if `T1` is "pointer to member of `C1` of type *cv1* `U1`" and `T2` is "pointer to member of `C2` of type *cv2* `U2`" where `C1` is reference-related to `C2` or `C2` is reference-related to `C1` (8.2156.1), the cv-combined type of `T2` and `T1` or the cv-combined type of `T1` and `T2`, respectively;

— if `T1` and `T2` are similar multi-level mixed pointer and pointer to member types (4.246), the cv-combined type of `T1` and `T2`;

— otherwise, a program that necessitates the determination of a composite pointer type is ill-formed.

[*Example:*

```
typedef void *p;
typedef const int *q;
typedef int **pi;
typedef const int **pci;
```

The composite pointer type of `p` and `q` is "pointer to `const void`"; the composite pointer type of `pi` and `pci` is "pointer to `const` pointer to `const int`". — *end example*]

5.398 Primary expressions [expr.prim]

5.398.1 General [expr.prim.general]

primary-expression:
- *literal*
- `this`
- `(` *expression* `)`
- *id-expression*
- *lambda-expression*

id-expression:
- *unqualified-id*
- *qualified-id*

unqualified-id:
- *identifier*
- *operator-function-id*
- *conversion-function-id*
- *literal-operator-id*
- `~` *class-name*
- `~` *decltype-specifier*
- *template-id*

1

A *literal* is a primary expression. Its type depends on its form (2.299). A string literal is an lvalue; all other literals are prvalues.

1

The keyword `this` names a pointer to the object for which a non-static member function (9.508.1) is invoked or a non-static data member's initializer (9.194) is evaluated.

1

If a declaration declares a member function or member function template of a class `X`, the expression `this` is a prvalue of type "pointer to *cv-qualifier-seq* `X`" between the optional *cv-qualifer-seq* and the end of the *function-definition*, *member-declarator*, or *declarator*. It shall not appear before the optional *cv-qualifier-seq* and it shall not appear within the declaration of a static member function (although its type and value category are defined within a static member function as they are within a non-static member function). [*Note:* this is because declaration matching does not occur until the complete declarator is known. — *end note*] Unlike the object expression in other contexts, `*this` is not required to be of complete type for purposes of class member access (5.1194.1) outside the member function body. [*Note:* only class members declared prior to the declaration are visible. — *end note*] [*Example:*

```
struct A {
  char g();
  template<class T> auto f(T t) -> decltype(t + g())
    { return t + g(); }
};
template auto A::f(int t) -> decltype(t + g());
```

— *end example*]

1

Otherwise, if a *member-declarator* declares a non-static data member (9.194) of a class `X`, the expression `this` is a prvalue of type "pointer to `X`" within the optional *brace-or-equal-initializer*. It shall not appear elsewhere in the *member-declarator*.

1

The expression `this` shall not appear in any other context. [*Example:*

```
class Outer {
  int a[sizeof(*this)];                // error: not inside a member function
  unsigned int sz = sizeof(*this);     // OK: in brace-or-equal-initializer

  void f() {
    int b[sizeof(*this)];              // OK

    struct Inner {
      int c[sizeof(*this)];            // error: not inside a member function of Inner
    };
  }
};
```

— *end example*]

1

A parenthesized expression is a primary expression whose type and value are identical to those of the enclosed expression. The presence of parentheses does not affect whether the expression is an lvalue. The parenthesized expression can be used in exactly the same contexts as those where the enclosed expression can be used, and with the same meaning, except as otherwise indicated.

1 An *id-expression* is a restricted form of a *primary-expression*. [*Note:* an *id-expression* can appear after `.` and `->` operators (5.1194.1). — *end note*]

2 An *identifier* is an *id-expression* provided it has been suitably declared (Clause 7). [*Note:* for *operator-function-id*s, see 13.1122; for *conversion-function-id*s, see 12.742.1; for *literal-operator-id*s, see 13.1241.1; for *template-id*s, see 14.224. A *class-name* or *decltype-specifier* prefixed by `~` denotes a destructor; see 12.807. Within the definition of a non-static member function, an *identifier* that names a non-static member is transformed to a class member access expression (9.465.1). — *end note*] The type of the expression is the type of the *identifier*. The result is the entity denoted by the identifier. The result is an lvalue if the entity is a function, variable, or data member and a prvalue otherwise.

qualified-id:
  nested-name-specifier `template`$_{opt}$ *unqualified-id*
nested-name-specifier:
  `::`
  type-name `::`
  namespace-name `::`
  decltype-specifier `::`
  nested-name-specifier identifier `::`
  nested-name-specifier `template`$_{opt}$ *simple-template-id* `::`

The type denoted by a *decltype-specifier* in a *nested-name-specifier* shall be a class or enumeration type.

1

A *nested-name-specifier* that denotes a class, optionally followed by the keyword `template` (14.224), and then followed by the name of a member of either that class (9.194) or one of its base classes (Clause 10), is a *qualified-id*; 3.1282.0.1 describes name lookup for class members that appear in *qualified-ids*. The result is the member. The type of the result is the type of the member. The result is an lvalue if the member is a static member function or a data member and a prvalue otherwise. [*Note:* a class member can be referred to using a *qualified-id* at any point in its potential scope (3.760.1). — *end note*] Where *class-name* `::~` *class-name* is used, the two *class-name*s shall refer to the same class; this notation names the destructor (12.807). The form `~` *decltype-specifier* also denotes the destructor, but it shall not be used as the *unqualified-id* in a *qualified-id*. [*Note:* a *typedef-name* that names a class is a *class-name* (9.172). — *end note*]

1 A `::`, or a *nested-name-specifier* that names a namespace (7.1142), in either case followed by the name of a member of that namespace (or the name of a member of a namespace made visible by a *using-directive*) is a *qualified-id*; 3.1363.0.1 describes name lookup for namespace members that appear in *qualified-ids*. The result is the member. The type of the result is the type of the member. The result is an lvalue if the member is a function or a variable and a prvalue otherwise.

2

A *nested-name-specifier* that denotes an enumeration (7.1001), followed by the name of an enumerator of that enumeration, is a *qualified-id* that refers to the enumerator. The result is the enumerator. The type of the result is the type of the enumeration. The result is a prvalue.

1 In a *qualified-id*, if the *unqualified-id* is a *conversion-function-id*, its *conversion-type-id* shall denote the same type in both the context in which the entire *qualified-id* occurs and in the context of the class denoted by the *nested-name-specifier*.

2

An *id-expression* that denotes a non-static data member or non-static member function of a class can only be used:

— as part of a class member access (5.1194.1) in which the object expression refers to the member's class[63] or a class derived from that class, or

— to form a pointer to member (5.1898.1), or

— if that *id-expression* denotes a non-static data member and it appears in an unevaluated operand. [*Example:*

```
struct S {
  int m;
};
int i = sizeof(S::m);          // OK
int j = sizeof(S::m + 42);     // OK
```

63) This also applies when the object expression is an implicit `(*this)` (9.465.1).

— *end example*]

5.509.1 Lambda expressions [expr.prim.lambda]

1

Lambda expressions provide a concise way to create simple function objects. [*Example:*

```
#include <algorithm>
#include <cmath>
void abssort(float* x, unsigned N) {
  std::sort(x, x + N,
    [](float a, float b) {
      return std::abs(a) < std::abs(b);
    });
}
```

— *end example*]

lambda-expression:
  lambda-introducer lambda-declarator$_{opt}$ *compound-statement*

lambda-introducer:
  `[` *lambda-capture*$_{opt}$`]`

lambda-capture:
  capture-default
  capture-list
  capture-default `,` *capture-list*

capture-default:
  `&`
  `=`

capture-list:
  capture `...`$_{opt}$
  capture-list `,` *capture* `...`$_{opt}$

capture:
  simple-capture
  init-capture

simple-capture:
  identifier
  `&` *identifier*
  `this`

init-capture:
  identifier initializer
  `&` *identifier initializer*

lambda-declarator:
  `(` *parameter-declaration-clause* `)` `mutable`$_{opt}$
    exception-specification$_{opt}$ *attribute-specifier-seq*$_{opt}$ *trailing-return-type*$_{opt}$

1

The evaluation of a *lambda-expression* results in a prvalue temporary (12.306). This temporary is called the *closure object*. A *lambda-expression* shall not appear in an unevaluated operand (Clause 5), in a *template-argument*, in an *alias-declaration*, in a typedef declaration, or in the declaration of a function or function template outside its function body and default arguments. [*Note:* The intention is to prevent lambdas from appearing in a signature. — *end note*] [*Note:* A closure object behaves like a function object (20.283). — *end note*]

1

The type of the *lambda-expression* (which is also the type of the closure object) is a unique, unnamed non-union class type — called the *closure type* — whose properties are described below. This class type is neither an aggregate (8.1817.1) nor a literal type (3.2955). The closure type is declared in the smallest block scope, class scope, or namespace scope that contains the corresponding *lambda-expression*. [*Note:* This determines the set of namespaces and classes associated with the closure type (3.1092.1). The parameter types of a *lambda-declarator* do not affect these associated namespaces and classes. — *end note*] An implementation may define the closure type differently from what is described below provided this does not alter the observable behavior of the program other than by changing:

— the size and/or alignment of the closure type,

— whether the closure type is trivially copyable (Clause 9),

— whether the closure type is a standard-layout class (Clause 9), or

— whether the closure type is a POD class (Clause 9).

An implementation shall not add members of rvalue reference type to the closure type.

1

If a *lambda-expression* does not include a *lambda-declarator*, it is as if the *lambda-declarator* were `()`. The lambda return type is `auto`, which is replaced by the *trailing-return-type* if provided and/or deduced from `return` statements as described in 7.832.0.1. [*Example:*

```
auto x1 = [](int i){ return i; };      // OK: return type is int
auto x2 = []{ return { 1, 2 }; };      // error: deducing return type from braced-init-list
int j;
auto x3 = []()->auto&& { return j; }; // OK: return type is int&
```

— *end example*]

1

The closure type for a non-generic *lambda-expression* has a public inline function call operator (13.1196.1) whose parameters and return type are described by the *lambda-expression*'s *parameter-declaration-clause* and *trailing-return-type* respectively. For a generic lambda, the closure type has a public inline function call operator member template (14.641.1) whose *template-parameter-list* consists of one invented type *template-parameter* for each occurrence of `auto` in the lambda's *parameter-declaration-clause*, in order of appearance. The invented type *template-parameter* is a parameter pack if the corresponding *parameter-declaration* declares a function parameter pack (8.360.1). The return type and function parameters of the function call operator template are derived from the *lambda-expression*'s *trailing-return-type* and *parameter-declaration-clause* by replacing each occurrence of `auto` in the *decl-specifier*s of the *parameter-declaration-clause* with the name of the corresponding invented *template-parameter*. [*Example:*

```
auto glambda = [](auto a, auto&& b) { return a < b; };
bool b = glambda(3, 3.14);                                  // OK
auto vglambda = [](auto printer) {
   return [=](auto&& ... ts) {                              // OK: ts is a function parameter pack
       printer(std::forward<decltype(ts)>(ts)...);

       return [=]() {
         printer(ts ...);
       };
   };
};
auto p = vglambda( [](auto v1, auto v2, auto v3)
                       { std::cout << v1 << v2 << v3; } );
```

```
auto q = p(1, 'a', 3.14);                    // OK: outputs 1a3.14
q();                                         // OK: outputs 1a3.14
```

— *end example*] This function call operator or operator template is declared `const` (9.465.1) if and only if the *lambda-expression*'s *parameter-declaration-clause* is not followed by `mutable`. It is neither virtual nor declared `volatile`. Any *exception-specification* specified on a *lambda-expression* applies to the corresponding function call operator or operator template. An *attribute-specifier-seq* in a *lambda-declarator* appertains to the type of the corresponding function call operator or operator template. [*Note:* Names referenced in the *lambda-declarator* are looked up in the context in which the *lambda-expression* appears. — *end note*]

1

The closure type for a non-generic *lambda-expression* with no *lambda-capture* has a public non-virtual non-explicit const conversion function to pointer to function with C++ language linkage (7.1477) having the same parameter and return types as the closure type's function call operator. The value returned by this conversion function shall be the address of a function that, when invoked, has the same effect as invoking the closure type's function call operator. For a generic lambda with no *lambda-capture*, the closure type has a public non-virtual non-explicit const conversion function template to pointer to function. The conversion function template has the same invented *template-parameter-list*, and the pointer to function has the same parameter types, as the function call operator template. The return type of the pointer to function shall behave as if it were a *decltype-specifier* denoting the return type of the corresponding function call operator template specialization. [*Note:* If the generic lambda has no *trailing-return-type* or the *trailing-return-type* contains a placeholder type, return type deduction of the corresponding function call operator template specialization has to be done. The corresponding specialization is that instantiation of the function call operator template with the same template arguments as those deduced for the conversion function template. Consider the following:

```
auto glambda = [](auto a) { return a; };
int (*fp)(int) = glambda;
```

The behavior of the conversion function of `glambda` above is like that of the following conversion function:

```
struct Closure {
  template<class T> auto operator()(T t) const { ... }
  template<class T> static auto lambda_call_operator_invoker(T a) {
    // forwards execution to operator()(a) and therefore has
    // the same return type deduced
    ...
  }
  template<class T> using fptr_t =
     decltype(lambda_call_operator_invoker(declval<T>())) (*)(T);

  template<class T> operator fptr_t<T>() const
    { return &lambda_call_operator_invoker; }
};
```

— *end note*] [*Example:*

```
void f1(int (*)(int))   { }
void f2(char (*)(int))  { }

void g(int (*)(int))    { }  // #1
void g(char (*)(char))  { }  // #2

void h(int (*)(int))    { }  // #3
```

```
void h(char (*)(int))   { }  // #4

auto glambda = [](auto a) { return a; };
f1(glambda);  // OK
f2(glambda);  // error: ID is not convertible
g(glambda);   // error: ambiguous
h(glambda);   // OK: calls #3 since it is convertible from ID
int& (*fpi)(int*) = [](auto* a) -> auto& { return *a; }; // OK
```

— *end example*] The value returned by any given specialization of this conversion function template shall be the address of a function that, when invoked, has the same effect as invoking the generic lambda's corresponding function call operator template specialization. [*Note:* This will result in the implicit instantiation of the generic lambda's body. The instantiated generic lambda's return type and parameter types shall match the return type and parameter types of the pointer to function. — *end note*] [*Example:*

```
auto GL = [](auto a) { std::cout << a; return a; };
int (*GL_int)(int) = GL;  // OK: through conversion function template
GL_int(3);                // OK: same as GL(3)
```

— *end example*]

1

The *lambda-expression*'s *compound-statement* yields the *function-body* (8.706) of the function call operator, but for purposes of name lookup (3.923), determining the type and value of `this` (9.508.1) and transforming *id-expressions* referring to non-static class members into class member access expressions using `(*this)` (9.465.1), the *compound-statement* is considered in the context of the *lambda-expression*. [*Example:*

```
struct S1 {
  int x, y;
  int operator()(int);
  void f() {
    [=]()->int {
      return operator()(this->x + y); // equivalent to S1::operator()(this->x + (*this).y)
                                      // this has type S1*
    };
  }
};
```

— *end example*] Further, a variable `__func__` is implicitly defined at the beginning of the *compound-statement* of the *lambda-expression*, with semantics as described in 8.706.1.

1

If a *lambda-capture* includes a *capture-default* that is `&`, no identifier in a *simple-capture* of that *lambda-capture* shall be preceded by `&`. If a *lambda-capture* includes a *capture-default* that is `=`, each *simple-capture* of that *lambda-capture* shall be of the form "`&` *identifier*". Ignoring appearances in *initializer*s of *init-capture*s, an identifier or `this` shall not appear more than once in a *lambda-capture*. [*Example:*

```
struct S2 { void f(int i); };
void S2::f(int i) {
  [&, i]{ };     // OK
  [&, &i]{ };    // error: i preceded by & when & is the default
  [=, this]{ }; // error: this when = is the default
  [i, i]{ };     // error: i repeated
}
```

— *end example*]

1

A *lambda-expression* whose smallest enclosing scope is a block scope (3.694.1) is a *local lambda expression*; any other *lambda-expression* shall not have a *capture-default* or *simple-capture* in its *lambda-introducer*. The *reaching scope* of a local lambda expression is the set of enclosing scopes up to and including the innermost enclosing function and its parameters. [*Note:* This reaching scope includes any intervening *lambda-expressions*. — *end note*]

1

The *identifier* in a *simple-capture* is looked up using the usual rules for unqualified name lookup (3.952.1); each such lookup shall find an entity. An entity that is designated by a *simple-capture* is said to be *explicitly captured*, and shall be `this` or a variable with automatic storage duration declared in the reaching scope of the local lambda expression.

1

An *init-capture* behaves as if it declares and explicitly captures a variable of the form "`auto` *init-capture* `;`" whose declarative region is the *lambda-expression*'s *compound-statement*, except that:

— if the capture is by copy (see below), the non-static data member declared for the capture and the variable are treated as two different ways of referring to the same object, which has the lifetime of the non-static data member, and no additional copy and destruction is performed, and

— if the capture is by reference, the variable's lifetime ends when the closure object's lifetime ends.

[*Note:* This enables an *init-capture* like "`x = std::move(x)`"; the second "`x`" must bind to a declaration in the surrounding context. — *end note*] [*Example:*

```
int x = 4;
auto y = [&r = x, x = x+1]()->int {
            r += 2;
            return x+2;
          }();  // Updates ::x to 6, and initializes y to 7.
```

— *end example*]

1

A *lambda-expression* with an associated *capture-default* that does not explicitly capture `this` or a variable with automatic storage duration (this excludes any *id-expression* that has been found to refer to an *init-capture*'s associated non-static data member), is said to *implicitly capture* the entity (i.e., `this` or a variable) if the *compound-statement*:

— odr-uses (3.91) the entity, or

— names the entity in a potentially-evaluated expression (3.91) where the enclosing full-expression depends on a generic lambda parameter declared within the reaching scope of the *lambda-expression*.

[*Example:*

```
void f(int, const int (&)[2] = {})    { }   // #1
void f(const int&, const int (&)[1])  { }   // #2
void test() {
  const int x = 17;
  auto g = [](auto a) {
    f(x);  // OK: calls #1, does not capture x
  };
```

```
  auto g2 = [=](auto a) {
    int selector[sizeof(a) == 1 ? 1 : 2]{};
    f(x, selector);  // OK: is a dependent expression, so captures x
  };
}
```

— *end example*] All such implicitly captured entities shall be declared within the reaching scope of the lambda expression. [*Note:* The implicit capture of an entity by a nested *lambda-expression* can cause its implicit capture by the containing *lambda-expression* (see below). Implicit odr-uses of `this` can result in implicit capture. — *end note*]

1

An entity is *captured* if it is captured explicitly or implicitly. An entity captured by a *lambda-expression* is odr-used (3.91) in the scope containing the *lambda-expression*. If `this` is captured by a local lambda expression, its nearest enclosing function shall be a non-static member function. If a *lambda-expression* or an instantiation of the function call operator template of a generic lambda odr-uses (3.91) `this` or a variable with automatic storage duration from its reaching scope, that entity shall be captured by the *lambda-expression*. If a *lambda-expression* captures an entity and that entity is not defined or captured in the immediately enclosing lambda expression or function, the program is ill-formed. [*Example:*

```
void f1(int i) {
  int const N = 20;
  auto m1 = [=]{
    int const M = 30;
    auto m2 = [i]{
      int x[N][M];              // OK: N and M are not odr-used
      x[0][0] = i;              // OK: i is explicitly captured by m2
                                // and implicitly captured by m1
    };
  };
  struct s1 {
    int f;
    void work(int n) {
      int m = n*n;
      int j = 40;
      auto m3 = [this,m] {
        auto m4 = [&,j] {       // error: j not captured by m3
          int x = n;            // error: n implicitly captured by m4
                                // but not captured by m3
          x += m;               // OK: m implicitly captured by m4
                                // and explicitly captured by m3
          x += i;               // error: i is outside of the reaching scope
          x += f;               // OK: this captured implicitly by m4
                                // and explicitly by m3
        };
      };
    }
  };
}
```

— *end example*]

1

A *lambda-expression* appearing in a default argument shall not implicitly or explicitly capture any entity.

[*Example:*

```
void f2() {
  int i = 1;
  void g1(int = ([i]{ return i; })());        // ill-formed
  void g2(int = ([i]{ return 0; })());        // ill-formed
  void g3(int = ([=]{ return i; })());        // ill-formed
  void g4(int = ([=]{ return 0; })());        // OK
  void g5(int = ([]{ return sizeof i; })());  // OK
}
```

— *end example*]

1

An entity is *captured by copy* if it is implicitly captured and the *capture-default* is `=` or if it is explicitly captured with a capture that is not of the form `&` *identifier* or `&` *identifier initializer*. For each entity captured by copy, an unnamed non-static data member is declared in the closure type. The declaration order of these members is unspecified. The type of such a data member is the type of the corresponding captured entity if the entity is not a reference to an object, or the referenced type otherwise. [*Note:* If the captured entity is a reference to a function, the corresponding data member is also a reference to a function. — *end note*] A member of an anonymous union shall not be captured by copy.

1

An entity is *captured by reference* if it is implicitly or explicitly captured but not captured by copy. It is unspecified whether additional unnamed non-static data members are declared in the closure type for entities captured by reference. A member of an anonymous union shall not be captured by reference.

1 If a *lambda-expression* `m2` captures an entity and that entity is captured by an immediately enclosing *lambda-expression* `m1`, then `m2`'s capture is transformed as follows:

(1.1) — if `m1` captures the entity by copy, `m2` captures the corresponding non-static data member of `m1`'s closure type;

(1.2) — if `m1` captures the entity by reference, `m2` captures the same entity captured by `m1`.

[*Example:* the nested lambda expressions and invocations below will output `123234`.

```
int a = 1, b = 1, c = 1;
auto m1 = [a, &b, &c]() mutable {
  auto m2 = [a, b, &c]() mutable {
    std::cout << a << b << c;
    a = 4; b = 4; c = 4;
  };
  a = 3; b = 3; c = 3;
  m2();
};
a = 2; b = 2; c = 2;
m1();
std::cout << a << b << c;
```

— *end example*]

2

Every *id-expression* within the *compound-statement* of a *lambda-expression* that is an odr-use (3.91) of an entity captured by copy is transformed into an access to the corresponding unnamed data member of the closure type. [*Note:* An *id-expression* that is not an odr-use refers to the original entity, never to a member of the closure type. Furthermore, such an *id-expression* does not cause the implicit capture of the entity.

— *end note*] If `this` is captured, each odr-use of `this` is transformed into an access to the corresponding unnamed data member of the closure type, cast (5.2765) to the type of `this`. [*Note:* The cast ensures that the transformed expression is a prvalue. — *end note*] [*Example:*

```
void f(const int*);
void g() {
  const int N = 10;
  [=] {
    int arr[N];              // OK: not an odr-use, refers to automatic variable
    f(&N);                   // OK: causes N to be captured; &N points to the
                             // corresponding member of the closure type
  };
}
```

— *end example*]

1

Every occurrence of `decltype((x))` where `x` is a possibly parenthesized *id-expression* that names an entity of automatic storage duration is treated as if `x` were transformed into an access to a corresponding data member of the closure type that would have been declared if `x` were an odr-use of the denoted entity. [*Example:*

```
void f3() {
  float x, &r = x;
  [=] {                      // x and r are not captured (appearance in a decltype operand is not an odr-use)
    decltype(x) y1;          // y1 has type float
    decltype((x)) y2 = y1;   // y2 has type float const& because this lambda
                             // is not mutable and x is an lvalue
    decltype(r) r1 = y1;     // r1 has type float& (transformation not considered)
    decltype((r)) r2 = y2;   // r2 has type float const&
  };
}
```

— *end example*]

1

The closure type associated with a *lambda-expression* has a deleted (8.793.1) default constructor and a deleted copy assignment operator. It has an implicitly-declared copy constructor (12.1577) and may have an implicitly-declared move constructor (12.1577). [*Note:* The copy/move constructor is implicitly defined in the same way as any other implicitly declared copy/move constructor would be implicitly defined. — *end note*]

1 The closure type associated with a *lambda-expression* has an implicitly-declared destructor (12.807).

2

When the *lambda-expression* is evaluated, the entities that are captured by copy are used to direct-initialize each corresponding non-static data member of the resulting closure object, and the non-static data members corresponding to the *init-capture*s are initialized as indicated by the corresponding *initializer* (which may be copy- or direct-initialization). (For array members, the array elements are direct-initialized in increasing subscript order.) These initializations are performed in the (unspecified) order in which the non-static data members are declared. [*Note:* This ensures that the destructions will occur in the reverse order of the constructions. — *end note*]

1

[*Note:* If an entity is implicitly or explicitly captured by reference, invoking the function call operator of the corresponding *lambda-expression* after the lifetime of the entity has ended is likely to result in undefined

behavior. — *end note*]

1

A *simple-capture* followed by an ellipsis is a pack expansion (14.703.1). An *init-capture* followed by an ellipsis is ill-formed. [*Example:*

```
template<class... Args>
void f(Args... args) {
  auto lm = [&, args...] { return g(args...); };
  lm();
}
```

— *end example*]

5.892 Postfix expressions [expr.post]

1

Postfix expressions group left-to-right.

postfix-expression:
- *primary-expression*
- *postfix-expression* `[` *expression* `]`
- *postfix-expression* `[` *braced-init-list* `]`
- *postfix-expression* `(` *expression-list*$_{opt}$ `)`
- *simple-type-specifier* `(` *expression-list*$_{opt}$ `)`
- *typename-specifier* `(` *expression-list*$_{opt}$ `)`
- *simple-type-specifier braced-init-list*
- *typename-specifier braced-init-list*
- *postfix-expression* `.` `template`$_{opt}$ *id-expression*
- *postfix-expression* `->` `template`$_{opt}$ *id-expression*
- *postfix-expression* `.` *pseudo-destructor-name*
- *postfix-expression* `->` *pseudo-destructor-name*
- *postfix-expression* `++`
- *postfix-expression* `--`
- `dynamic_cast` `<` *type-id* `>` `(` *expression* `)`
- `static_cast` `<` *type-id* `>` `(` *expression* `)`
- `reinterpret_cast` `<` *type-id* `>` `(` *expression* `)`
- `const_cast` `<` *type-id* `>` `(` *expression* `)`
- `typeid` `(` *expression* `)`
- `typeid` `(` *type-id* `)`

expression-list:
- *initializer-list*

pseudo-destructor-name:
- *nested-name-specifier*$_{opt}$ *type-name* `::` `~` *type-name*
- *nested-name-specifier* `template` *simple-template-id* `::` `~` *type-name*
- *nested-name-specifier*$_{opt}$ `~` *type-name*
- `~` *decltype-specifier*

1 [*Note:* The `>` token following the *type-id* in a `dynamic_cast`, `static_cast`, `reinterpret_cast`, or `const_cast` may be the product of replacing a `>>` token by two consecutive `>` tokens (14.224). — *end note*]

5.896.1 Subscripting [expr.sub]

1

A postfix expression followed by an expression in square brackets is a postfix expression. One of the expressions shall have the type "array of `T`" or "pointer to `T`" and the other shall have unscoped enumeration

or integral type. The result is of type "`T`." The type "`T`" shall be a completely-defined object type.[64] The expression `E1[E2]` is identical (by definition) to `*((E1)+(E2))` [*Note:* see 5.1897 and 5.2882 for details of `*` and `+` and 8.251.1 for details of arrays. — *end note*], except that in the case of an array operand, the result is an lvalue if that operand is an lvalue and an xvalue otherwise.

1

A *braced-init-list* shall not be used with the built-in subscript operator.

5.955.1 Function call [expr.call]

1

A function call is a postfix expression followed by parentheses containing a possibly empty, comma-separated list of *initializer-clause*s which constitute the arguments to the function. The postfix expression shall have function type or pointer to function type. For a call to a non-member function or to a static member function, the postfix expression shall be either an lvalue that refers to a function (in which case the function-to-pointer standard conversion (4.220) is suppressed on the postfix expression), or it shall have pointer to function type. Calling a function through an expression whose function type has a language linkage that is different from the language linkage of the function type of the called function's definition is undefined (7.1477). For a call to a non-static member function, the postfix expression shall be an implicit (9.465.1, 9.559) or explicit class member access (5.1194.1) whose *id-expression* is a function member name, or a pointer-to-member expression (5.2832) selecting a function member; the call is as a member of the class object referred to by the object expression. In the case of an implicit class member access, the implied object is the one pointed to by `this`. [*Note:* a member function call of the form `f()` is interpreted as `(*this).f()` (see 9.465.1). — *end note*] If a function or member function name is used, the name can be overloaded (Clause 13), in which case the appropriate function shall be selected according to the rules in 13.88. If the selected function is non-virtual, or if the *id-expression* in the class member access expression is a *qualified-id*, that function is called. Otherwise, its final overrider (10.113) in the dynamic type of the object expression is called; such a call is referred to as a *virtual function call*. [*Note:* the dynamic type is the type of the object referred to by the current value of the object expression. 12.1483 describes the behavior of virtual function calls when the object expression refers to an object under construction or destruction. — *end note*]

1

[*Note:* If a function or member function name is used, and name lookup (3.923) does not find a declaration of that name, the program is ill-formed. No function is implicitly declared by such a call. — *end note*]

1

If the *postfix-expression* designates a destructor (12.807), the type of the function call expression is `void`; otherwise, the type of the function call expression is the return type of the statically chosen function (i.e., ignoring the `virtual` keyword), even if the type of the function actually called is different. This return type shall be an object type, a reference type or *cv* `void`.

1

When a function is called, each parameter (8.360.1) shall be initialized (8.821, 12.1577, 12.30) with its corresponding argument. [*Note:* Such initializations are indeterminately sequenced with respect to each other (1.430) — *end note*] If the function is a non-static member function, the `this` parameter of the function (9.508.1) shall be initialized with a pointer to the object of the call, converted as if by an explicit type conversion (5.2765). [*Note:* There is no access or ambiguity checking on this conversion; the access checking and disambiguation are done as part of the (possibly implicit) class member access operator. See 10.52, 11.43, and 5.1194.1. — *end note*] When a function is called, the parameters that have object type shall have completely-defined object type. [*Note:* this still allows a parameter to be a pointer or reference to an incomplete class type. However, it prevents a passed-by-value parameter to have an incomplete class

64) This is true even if the subscript operator is used in the following common idiom: `&x[0]`.

type. — *end note*] During the initialization of a parameter, an implementation may avoid the construction of extra temporaries by combining the conversions on the associated argument and/or the construction of temporaries with the initialization of the parameter (see 12.306). The lifetime of a parameter ends when the function in which it is defined returns. The initialization and destruction of each parameter occurs within the context of the calling function. [*Example:* the access of the constructor, conversion functions or destructor is checked at the point of call in the calling function. If a constructor or destructor for a function parameter throws an exception, the search for a handler starts in the scope of the calling function; in particular, if the function called has a *function-try-block* (Clause 15) with a handler that could handle the exception, this handler is not considered. — *end example*] The value of a function call is the value returned by the called function except in a virtual function call if the return type of the final overrider is different from the return type of the statically chosen function, the value returned from the final overrider is converted to the return type of the statically chosen function.

1

[*Note:* a function can change the values of its non-const parameters, but these changes cannot affect the values of the arguments except where a parameter is of a reference type (8.97.1); if the reference is to a const-qualified type, `const_cast` is required to be used to cast away the constness in order to modify the argument's value. Where a parameter is of `const` reference type a temporary object is introduced if needed (7.519.1, 2.299, 2.453.1, 8.251.1, 12.306). In addition, it is possible to modify the values of nonconstant objects through pointer parameters. — *end note*]

1

A function can be declared to accept fewer arguments (by declaring default arguments (8.545.1)) or more arguments (by using the ellipsis, `...`, or a function parameter pack (8.360.1)) than the number of parameters in the function definition (8.706). [*Note:* this implies that, except where the ellipsis (`...`) or a function parameter pack is used, a parameter is available for each argument. — *end note*]

1

When there is no parameter for a given argument, the argument is passed in such a way that the receiving function can obtain the value of the argument by invoking `va_arg` (18.295). [*Note:* This paragraph does not apply to arguments passed to a function parameter pack. Function parameter packs are expanded during template instantiation (14.703.1), thus each such argument has a corresponding parameter when a function template specialization is actually called. — *end note*] The lvalue-to-rvalue (4.127), array-to-pointer (4.180), and function-to-pointer (4.220) standard conversions are performed on the argument expression. An argument that has (possibly cv-qualified) type `std::nullptr_t` is converted to type `void*` (4.579). After these conversions, if the argument does not have arithmetic, enumeration, pointer, pointer to member, or class type, the program is ill-formed. Passing a potentially-evaluated argument of class type (Clause 9) having a non-trivial copy constructor, a non-trivial move constructor, or a non-trivial destructor, with no corresponding parameter, is conditionally-supported with implementation-defined semantics. If the argument has integral or enumeration type that is subject to the integral promotions (4.262), or a floating point type that is subject to the floating point promotion (4.389), the value of the argument is converted to the promoted type before the call. These promotions are referred to as the *default argument promotions*.

1

[*Note:* The evaluations of the postfix expression and of the arguments are all unsequenced relative to one another. All side effects of argument evaluations are sequenced before the function is entered (see 1.430). — *end note*]

1

Recursive calls are permitted, except to the `main` function (3.1721.1).

1

A function call is an lvalue if the result type is an lvalue reference type or an rvalue reference to function type, an xvalue if the result type is an rvalue reference to object type, and a prvalue otherwise.

1

If a function call is a prvalue of object type:

— if the function call is either

— the operand of a *decltype-specifier* or

— the right operand of a comma operator that is the operand of a *decltype-specifier*,

a temporary object is not introduced for the prvalue. The type of the prvalue may be incomplete. [*Note:* as a result, storage is not allocated for the prvalue and it is not destroyed; thus, a class type is not instantiated as a result of being the type of a function call in this context. This is true regardless of whether the expression uses function call notation or operator notation (13.264.0.1). — *end note*] [*Note:* unlike the rule for a *decltype-specifier* that considers whether an *id-expression* is parenthesized (7.678.0.1), parentheses have no special meaning in this context. — *end note*]

— otherwise, the type of the prvalue shall be complete.

5.1108.1 Explicit type conversion (functional notation) [expr.type.conv]

1

A *simple-type-specifier* (7.678.0.1) or *typename-specifier* (14.1061) followed by a parenthesized *expression-list* constructs a value of the specified type given the expression list. If the expression list is a single expression, the type conversion expression is equivalent (in definedness, and if defined in meaning) to the corresponding cast expression (5.2765). If the type specified is a class type, the class type shall be complete. If the expression list specifies more than a single value, the type shall be a class with a suitably declared constructor (8.821, 12.30), and the expression `T(x1, x2, ...)` is equivalent in effect to the declaration `T t(x1, x2, ...);` for some invented temporary variable `t`, with the result being the value of `t` as a prvalue.

1

The expression `T()`, where `T` is a *simple-type-specifier* or *typename-specifier* for a non-array complete object type or the (possibly cv-qualified) `void` type, creates a prvalue of the specified type, whose value is that produced by value-initializing (8.821) an object of type `T`; no initialization is done for the `void()` case. [*Note:* if `T` is a non-class type that is cv-qualified, the *cv-qualifier*s are discarded when determining the type of the resulting prvalue (Clause 5). — *end note*]

1

Similarly, a *simple-type-specifier* or *typename-specifier* followed by a *braced-init-list* creates a temporary object of the specified type direct-list-initialized (8.2435.1) with the specified *braced-init-list*, and its value is that temporary object as a prvalue.

5.1181.1 Pseudo destructor call [expr.pseudo]

1

The use of a *pseudo-destructor-name* after a dot `.` or arrow `->` operator represents the destructor for the non-class type denoted by *type-name* or *decltype-specifier*. The result shall only be used as the operand for the function call operator `()`, and the result of such a call has type `void`. The only effect is the evaluation of the *postfix-expression* before the dot or arrow.

1

The left-hand side of the dot operator shall be of scalar type. The left-hand side of the arrow operator shall be of pointer to scalar type. This scalar type is the object type. The *cv*-unqualified versions of the object

type and of the type designated by the *pseudo-destructor-name* shall be the same type. Furthermore, the two *type-name*s in a *pseudo-destructor-name* of the form

nested-name-specifier$_{opt}$ *type-name* `::` `~` *type-name*

shall designate the same scalar type.

5.1194.1 Class member access [expr.ref]

1

A postfix expression followed by a dot `.` or an arrow `->`, optionally followed by the keyword `template` (14.224), and then followed by an *id-expression*, is a postfix expression. The postfix expression before the dot or arrow is evaluated;[65] the result of that evaluation, together with the *id-expression*, determines the result of the entire postfix expression.

1

For the first option (dot) the first expression shall have complete class type. For the second option (arrow) the first expression shall have pointer to complete class type. The expression `E1->E2` is converted to the equivalent form `(*(E1)).E2`; the remainder of 5.1194.1 will address only the first option (dot).[66] In either case, the *id-expression* shall name a member of the class or of one of its base classes. [*Note:* because the name of a class is inserted in its class scope (Clause 9), the name of a class is also considered a nested member of that class. — *end note*] [*Note:* 3.1423.1 describes how names are looked up after the `.` and `->` operators. — *end note*]

1

Abbreviating *postfix-expression.id-expression* as `E1.E2`, `E1` is called the *object expression*. The type and value category of `E1.E2` are determined as follows. In the remainder of 5.1194.1, *cq* represents either `const` or the absence of `const` and *vq* represents either `volatile` or the absence of `volatile`. *cv* represents an arbitrary set of cv-qualifiers, as defined in 3.3640.1.

1

If `E2` is declared to have type "reference to `T`," then `E1.E2` is an lvalue; the type of `E1.E2` is `T`. Otherwise, one of the following rules applies.

— If `E2` is a static data member and the type of `E2` is `T`, then `E1.E2` is an lvalue; the expression designates the named member of the class. The type of `E1.E2` is `T`.

— If `E2` is a non-static data member and the type of `E1` is "*cq1 vq1* `X`", and the type of `E2` is "*cq2 vq2* `T`", the expression designates the named member of the object designated by the first expression. If `E1` is an lvalue, then `E1.E2` is an lvalue; otherwise `E1.E2` is an xvalue. Let the notation *vq12* stand for the "union" of *vq1* and *vq2*; that is, if *vq1* or *vq2* is `volatile`, then *vq12* is `volatile`. Similarly, let the notation *cq12* stand for the "union" of *cq1* and *cq2*; that is, if *cq1* or *cq2* is `const`, then *cq12* is `const`. If `E2` is declared to be a `mutable` member, then the type of `E1.E2` is "*vq12* `T`". If `E2` is not declared to be a `mutable` member, then the type of `E1.E2` is "*cq12 vq12* `T`".

— If `E2` is a (possibly overloaded) member function, function overload resolution (13.88) is used to determine whether `E1.E2` refers to a static or a non-static member function.

  — If it refers to a static member function and the type of `E2` is "function of parameter-type-list returning `T`", then `E1.E2` is an lvalue; the expression designates the static member function. The type of `E1.E2` is the same type as that of `E2`, namely "function of parameter-type-list returning `T`".

65) If the class member access expression is evaluated, the subexpression evaluation happens even if the result is unnecessary to determine the value of the entire postfix expression, for example if the *id-expression* denotes a static member.

66) Note that `(*(E1))` is an lvalue.

— Otherwise, if `E1.E2` refers to a non-static member function and the type of `E2` is "function of parameter-type-list *cv ref-qualifier*$_{opt}$ returning `T`", then `E1.E2` is a prvalue. The expression designates a non-static member function. The expression can be used only as the left-hand operand of a member function call (9.401). [*Note:* Any redundant set of parentheses surrounding the expression is ignored (5.398). — *end note*] The type of `E1.E2` is "function of parameter-type-list *cv* returning `T`".

— If `E2` is a nested type, the expression `E1.E2` is ill-formed.

— If `E2` is a member enumerator and the type of `E2` is `T`, the expression `E1.E2` is a prvalue. The type of `E1.E2` is `T`.

1

If `E2` is a non-static data member or a non-static member function, the program is ill-formed if the class of which `E2` is directly a member is an ambiguous base (10.52) of the naming class (11.43) of `E2`. [*Note:* The program is also ill-formed if the naming class is an ambiguous base of the class type of the object expression; see 11.43. — *end note*]

5.1264.1 Increment and decrement [expr.post.incr]

1

The value of a postfix `++` expression is the value of its operand. [*Note:* the value obtained is a copy of the original value — *end note*] The operand shall be a modifiable lvalue. The type of the operand shall be an arithmetic type or a pointer to a complete object type. The value of the operand object is modified by adding `1` to it, unless the object is of type `bool`, in which case it is set to `true`. [*Note:* this use is deprecated, see Annex D. — *end note*] The value computation of the `++` expression is sequenced before the modification of the operand object. With respect to an indeterminately-sequenced function call, the operation of postfix `++` is a single evaluation. [*Note:* Therefore, a function call shall not intervene between the lvalue-to-rvalue conversion and the side effect associated with any single postfix ++ operator. — *end note*] The result is a prvalue. The type of the result is the cv-unqualified version of the type of the operand. See also 5.2882 and 5.3483.

1

The operand of postfix `--` is decremented analogously to the postfix `++` operator, except that the operand shall not be of type `bool`. [*Note:* For prefix increment and decrement, see 5.2054.1. — *end note*]

5.1295.1 Dynamic cast [expr.dynamic.cast]

1

The result of the expression `dynamic_cast<T>(v)` is the result of converting the expression `v` to type `T`. `T` shall be a pointer or reference to a complete class type, or "pointer to *cv* `void`." The `dynamic_cast` operator shall not cast away constness (5.1871.1).

1

If `T` is a pointer type, `v` shall be a prvalue of a pointer to complete class type, and the result is a prvalue of type `T`. If `T` is an lvalue reference type, `v` shall be an lvalue of a complete class type, and the result is an lvalue of the type referred to by `T`. If `T` is an rvalue reference type, `v` shall be an expression having a complete class type, and the result is an xvalue of the type referred to by `T`.

1

If the type of `v` is the same as `T`, or it is the same as `T` except that the class object type in `T` is more cv-qualified than the class object type in `v`, the result is `v` (converted if necessary).

1

If the value of `v` is a null pointer value in the pointer case, the result is the null pointer value of type `T`.

1

If `T` is "pointer to *cv1* `B`" and `v` has type "pointer to *cv2* `D`" such that `B` is a base class of `D`, the result is a pointer to the unique `B` subobject of the `D` object pointed to by `v`. Similarly, if `T` is "reference to *cv1* `B`" and `v` has type *cv2* `D` such that `B` is a base class of `D`, the result is the unique `B` subobject of the `D` object referred to by `v`. [67] The result is an lvalue if `T` is an lvalue reference, or an xvalue if `T` is an rvalue reference. In both the pointer and reference cases, the program is ill-formed if *cv2* has greater cv-qualification than *cv1* or if `B` is an inaccessible or ambiguous base class of `D`. [*Example:*

```
struct B { };
struct D : B { };
void foo(D* dp) {
  B*  bp = dynamic_cast<B*>(dp);     // equivalent to B* bp = dp;
}
```

— *end example*]

1

Otherwise, `v` shall be a pointer to or a glvalue of a polymorphic type (10.113).

1

If `T` is "pointer to *cv* `void`," then the result is a pointer to the most derived object pointed to by `v`. Otherwise, a run-time check is applied to see if the object pointed or referred to by `v` can be converted to the type pointed or referred to by `T`.

1

If `C` is the class type to which `T` points or refers, the run-time check logically executes as follows:

— If, in the most derived object pointed (referred) to by `v`, `v` points (refers) to a `public` base class subobject of a `C` object, and if only one object of type `C` is derived from the subobject pointed (referred) to by `v` the result points (refers) to that `C` object.

— Otherwise, if `v` points (refers) to a `public` base class subobject of the most derived object, and the type of the most derived object has a base class, of type `C`, that is unambiguous and `public`, the result points (refers) to the `C` subobject of the most derived object.

— Otherwise, the run-time check *fails*.

1

The value of a failed cast to pointer type is the null pointer value of the required result type. A failed cast to reference type throws an exception (15.7) of a type that would match a handler (15.146) of type `std::bad_cast` (18.240.1).

[*Example:*

```
class A { virtual void f(); };
class B { virtual void g(); };
class D : public virtual A, private B { };
void g() {
  D   d;
  B*  bp = (B*)&d;                  // cast needed to break protection
  A*  ap = &d;                      // public derivation, no cast needed
  D&  dr = dynamic_cast<D&>(*bp);   // fails
```

67) The most derived object (1.319) pointed or referred to by `v` can contain other `B` objects as base classes, but these are ignored.

```
  ap = dynamic_cast<A*>(bp);        // fails
  bp = dynamic_cast<B*>(ap);        // fails
  ap = dynamic_cast<A*>(&d);        // succeeds
  bp = dynamic_cast<B*>(&d);        // ill-formed (not a run-time check)
}

class E : public D, public B { };
class F : public E, public D { };
void h() {
  F   f;
  A*  ap  = &f;                     // succeeds: finds unique A
  D*  dp  = dynamic_cast<D*>(ap);   // fails: yields 0
                                    // f has two D subobjects
  E*  ep  = (E*)ap;                 // ill-formed: cast from virtual base
  E*  ep1 = dynamic_cast<E*>(ap);   // succeeds
}
```

— *end example*] [*Note:* 12.1483 describes the behavior of a `dynamic_cast` applied to an object under construction or destruction. — *end note*]

5.1393.1 Type identification [expr.typeid]

1

The result of a `typeid` expression is an lvalue of static type `const std::type_info` (18.227.1) and dynamic type `const std::type_info` or `const` *name* where *name* is an implementation-defined class publicly derived from `std::type_info` which preserves the behavior described in 18.227.1.[68] The lifetime of the object referred to by the lvalue extends to the end of the program. Whether or not the destructor is called for the `std::type_info` object at the end of the program is unspecified.

1

When `typeid` is applied to a glvalue expression whose type is a polymorphic class type (10.113), the result refers to a `std::type_info` object representing the type of the most derived object (1.319) (that is, the dynamic type) to which the glvalue refers. If the glvalue expression is obtained by applying the unary `*` operator to a pointer[69] and the pointer is a null pointer value (4.579), the `typeid` expression throws an exception (15.7) of a type that would match a handler of type `std::bad_typeid` exception (18.240.2).

1

When `typeid` is applied to an expression other than a glvalue of a polymorphic class type, the result refers to a `std::type_info` object representing the static type of the expression. Lvalue-to-rvalue (4.127), array-to-pointer (4.180), and function-to-pointer (4.220) conversions are not applied to the expression. If the type of the expression is a class type, the class shall be completely-defined. The expression is an unevaluated operand (Clause 5).

1

When `typeid` is applied to a *type-id*, the result refers to a `std::type_info` object representing the type of the *type-id*. If the type of the *type-id* is a reference to a possibly *cv*-qualified type, the result of the `typeid` expression refers to a `std::type_info` object representing the *cv*-unqualified referenced type. If the type of the *type-id* is a class type or a reference to a class type, the class shall be completely-defined.

1

If the type of the expression or *type-id* is a cv-qualified type, the result of the `typeid` expression refers to a `std::type_info` object representing the cv-unqualified type. [*Example:*

68) The recommended name for such a class is `extended_type_info`.

69) If `p` is an expression of pointer type, then `*p`, `(*p)`, `*(p)`, `((*p))`, `*((p))`, and so on all meet this requirement.

```
class D { /* ...  */ };
D d1;
const D d2;

typeid(d1) == typeid(d2);       // yields true
typeid(D)  == typeid(const D);  // yields true
typeid(D)  == typeid(d2);       // yields true
typeid(D)  == typeid(const D&); // yields true
```

— *end example*]

1

If the header `<typeinfo>` (18.227.1) is not included prior to a use of `typeid`, the program is ill-formed.

1 [*Note:* 12.1483 describes the behavior of `typeid` applied to an object under construction or destruction. — *end note*]

5.1443.1 Static cast [expr.static.cast]

1

The result of the expression `static_cast<T>(v)` is the result of converting the expression `v` to type `T`. If `T` is an lvalue reference type or an rvalue reference to function type, the result is an lvalue; if `T` is an rvalue reference to object type, the result is an xvalue; otherwise, the result is a prvalue. The `static_cast` operator shall not cast away constness (5.1871.1).

1

An lvalue of type "*cv1* `B`," where `B` is a class type, can be cast to type "reference to *cv2* `D`," where `D` is a class derived (Clause 10) from `B`, if a valid standard conversion from "pointer to `D`" to "pointer to `B`" exists (4.579), *cv2* is the same cv-qualification as, or greater cv-qualification than, *cv1*, and `B` is neither a virtual base class of `D` nor a base class of a virtual base class of `D`. The result has type "*cv2* `D`." An xvalue of type "*cv1* `B`" may be cast to type "rvalue reference to *cv2* `D`" with the same constraints as for an lvalue of type "*cv1* `B`." If the object of type "*cv1* `B`" is actually a subobject of an object of type `D`, the result refers to the enclosing object of type `D`. Otherwise, the behavior is undefined. [*Example:*

```
struct B { };
struct D : public B { };
D d;
B &br = d;

static_cast<D&>(br);            // produces lvalue to the original d object
```

— *end example*]

1

A glvalue, class prvalue, or array prvalue of type "*cv1* `T1`" can be cast to type "rvalue reference to *cv2* `T2`" if "*cv2* `T2`" is reference-compatible with "*cv1* `T1`" (8.2156.1). If the value is not a bit-field, the result refers to the object or the specified base class subobject thereof; otherwise, the lvalue-to-rvalue conversion (4.127) is applied to the bit-field and the resulting prvalue is used as the *expression* of the `static_cast` for the remainder of this section. If `T2` is an inaccessible (Clause 11) or ambiguous (10.52) base class of `T1`, a program that necessitates such a cast is ill-formed.

1

An expression `e` can be explicitly converted to a type `T` using a `static_cast` of the form `static_cast<T>(e)` if the declaration `T t(e);` is well-formed, for some invented temporary variable `t` (8.821). The effect of such an explicit conversion is the same as performing the declaration and initialization and then using the

temporary variable as the result of the conversion. The expression `e` is used as a glvalue if and only if the initialization uses it as a glvalue.

1

Otherwise, the `static_cast` shall perform one of the conversions listed below. No other conversion shall be performed explicitly using a `static_cast`.

1

Any expression can be explicitly converted to type *cv* `void`, in which case it becomes a discarded-value expression (Clause 5). [*Note:* however, if the value is in a temporary object (12.306), the destructor for that object is not executed until the usual time, and the value of the object is preserved for the purpose of executing the destructor. — *end note*]

1

The inverse of any standard conversion sequence (Clause 4) not containing an lvalue-to-rvalue (4.127), array-to-pointer (4.180), function-to-pointer (4.220), null pointer (4.579), null member pointer (4.710), or boolean (4.732) conversion, can be performed explicitly using `static_cast`. A program is ill-formed if it uses `static_cast` to perform the inverse of an ill-formed standard conversion sequence. [*Example:*

```
struct B { };
struct D : private B { };
void f() {
  static_cast<D*>((B*)0);                // Error: B is a private base of D.
  static_cast<int B::*>((int D::*)0);    // Error: B is a private base of D.
}
```

— *end example*]

1

The lvalue-to-rvalue (4.127), array-to-pointer (4.180), and function-to-pointer (4.220) conversions are applied to the operand. Such a `static_cast` is subject to the restriction that the explicit conversion does not cast away constness (5.1871.1), and the following additional rules for specific cases:

1

A value of a scoped enumeration type (7.1001) can be explicitly converted to an integral type. When that type is *cv* `bool`, the resulting value is `false` if the original value is zero and `true` for all other values. For the remaining integral types, the value is unchanged if the original value can be represented by the specified type. Otherwise, the resulting value is unspecified. A value of a scoped enumeration type can also be explicitly converted to a floating-point type; the result is the same as that of converting from the original value to the floating-point type.

1

A value of integral or enumeration type can be explicitly converted to an enumeration type. The value is unchanged if the original value is within the range of the enumeration values (7.1001). Otherwise, the resulting value is unspecified (and might not be in that range). A value of floating-point type can also be explicitly converted to an enumeration type. The resulting value is the same as converting the original value to the underlying type of the enumeration (4.532), and subsequently to the enumeration type.

1

A prvalue of type "pointer to *cv1* `B`," where `B` is a class type, can be converted to a prvalue of type "pointer to *cv2* `D`," where `D` is a class derived (Clause 10) from `B`, if a valid standard conversion from "pointer to `D`" to "pointer to `B`" exists (4.579), *cv2* is the same cv-qualification as, or greater cv-qualification than, *cv1*, and `B` is neither a virtual base class of `D` nor a base class of a virtual base class of `D`. The null pointer value (4.579) is converted to the null pointer value of the destination type. If the prvalue of type "pointer to *cv1* `B`" points

to a B that is actually a subobject of an object of type D, the resulting pointer points to the enclosing object of type D. Otherwise, the behavior is undefined.

1

A prvalue of type "pointer to member of D of type *cv1* T" can be converted to a prvalue of type "pointer to member of B" of type *cv2* T, where B is a base class (Clause 10) of D, if a valid standard conversion from "pointer to member of B of type T" to "pointer to member of D of type T" exists (4.710), and *cv2* is the same cv-qualification as, or greater cv-qualification than, *cv1*.[70] The null member pointer value (4.710) is converted to the null member pointer value of the destination type. If class B contains the original member, or is a base or derived class of the class containing the original member, the resulting pointer to member points to the original member. Otherwise, the behavior is undefined. [*Note:* although class B need not contain the original member, the dynamic type of the object with which indirection through the pointer to member is performed must contain the original member; see 5.2832. — *end note*]

1

A prvalue of type "pointer to *cv1* void" can be converted to a prvalue of type "pointer to *cv2* T," where T is an object type and *cv2* is the same cv-qualification as, or greater cv-qualification than, *cv1*. The null pointer value is converted to the null pointer value of the destination type. If the original pointer value represents the address A of a byte in memory and A satisfies the alignment requirement of T, then the resulting pointer value represents the same address as the original pointer value, that is, A. The result of any other such pointer conversion is unspecified. A value of type pointer to object converted to "pointer to *cv* void" and back, possibly with different cv-qualification, shall have its original value. [*Example:*

```
T* p1 = new T;
const T* p2 = static_cast<const T*>(static_cast<void*>(p1));
bool b = p1 == p2;  // b will have the value true.
```

— *end example*]

5.1644.1 Reinterpret cast [expr.reinterpret.cast]

1

The result of the expression `reinterpret_cast<T>(v)` is the result of converting the expression v to type T. If T is an lvalue reference type or an rvalue reference to function type, the result is an lvalue; if T is an rvalue reference to object type, the result is an xvalue; otherwise, the result is a prvalue and the lvalue-to-rvalue (4.127), array-to-pointer (4.180), and function-to-pointer (4.220) standard conversions are performed on the expression v. Conversions that can be performed explicitly using `reinterpret_cast` are listed below. No other conversion can be performed explicitly using `reinterpret_cast`.

1

The `reinterpret_cast` operator shall not cast away constness (5.1871.1). An expression of integral, enumeration, pointer, or pointer-to-member type can be explicitly converted to its own type; such a cast yields the value of its operand.

1

[*Note:* The mapping performed by `reinterpret_cast` might, or might not, produce a representation different from the original value. — *end note*]

1

A pointer can be explicitly converted to any integral type large enough to hold it. The mapping function is implementation-defined. [*Note:* It is intended to be unsurprising to those who know the addressing structure of the underlying machine. — *end note*] A value of type `std::nullptr_t` can be converted to an integral

70) Function types (including those used in pointer to member function types) are never cv-qualified; see 8.360.1.

type; the conversion has the same meaning and validity as a conversion of `(void*)0` to the integral type. [*Note:* A `reinterpret_cast` cannot be used to convert a value of any type to the type `std::nullptr_t`. — *end note*]

1

A value of integral type or enumeration type can be explicitly converted to a pointer. A pointer converted to an integer of sufficient size (if any such exists on the implementation) and back to the same pointer type will have its original value; mappings between pointers and integers are otherwise implementation-defined. [*Note:* Except as described in 3.2532.0.1, the result of such a conversion will not be a safely-derived pointer value. — *end note*]

1

A function pointer can be explicitly converted to a function pointer of a different type. The effect of calling a function through a pointer to a function type (8.360.1) that is not the same as the type used in the definition of the function is undefined. Except that converting a prvalue of type "pointer to `T1`" to the type "pointer to `T2`" (where `T1` and `T2` are function types) and back to its original type yields the original pointer value, the result of such a pointer conversion is unspecified. [*Note:* see also 4.579 for more details of pointer conversions. — *end note*]

1

An object pointer can be explicitly converted to an object pointer of a different type.[71] When a prvalue `v` of object pointer type is converted to the object pointer type "pointer to *cv* `T`", the result is `static_cast<`*cv* `T*>(static_cast<`*cv* `void*>(v))`. Converting a prvalue of type "pointer to `T1`" to the type "pointer to `T2`" (where `T1` and `T2` are object types and where the alignment requirements of `T2` are no stricter than those of `T1`) and back to its original type yields the original pointer value.

1

Converting a function pointer to an object pointer type or vice versa is conditionally-supported. The meaning of such a conversion is implementation-defined, except that if an implementation supports conversions in both directions, converting a prvalue of one type to the other type and back, possibly with different cv-qualification, shall yield the original pointer value.

1

The null pointer value (4.579) is converted to the null pointer value of the destination type. [*Note:* A null pointer constant of type `std::nullptr_t` cannot be converted to a pointer type, and a null pointer constant of integral type is not necessarily converted to a null pointer value. — *end note*]

1

A prvalue of type "pointer to member of `X` of type `T1`" can be explicitly converted to a prvalue of a different type "pointer to member of `Y` of type `T2`" if `T1` and `T2` are both function types or both object types.[72] The null member pointer value (4.710) is converted to the null member pointer value of the destination type. The result of this conversion is unspecified, except in the following cases:

— converting a prvalue of type "pointer to member function" to a different pointer to member function type and back to its original type yields the original pointer to member value.

— converting a prvalue of type "pointer to data member of `X` of type `T1`" to the type "pointer to data member of `Y` of type `T2`" (where the alignment requirements of `T2` are no stricter than those of `T1`) and back to its original type yields the original pointer to member value.

71) The types may have different *cv*-qualifiers, subject to the overall restriction that a `reinterpret_cast` cannot cast away constness.

72) `T1` and `T2` may have different *cv*-qualifiers, subject to the overall restriction that a `reinterpret_cast` cannot cast away constness.

1

A glvalue expression of type `T1` can be cast to the type "reference to `T2`" if an expression of type "pointer to `T1`" can be explicitly converted to the type "pointer to `T2`" using a `reinterpret_cast`. The result refers to the same object as the source glvalue, but with the specified type. [*Note:* That is, for lvalues, a reference cast `reinterpret_cast<T&>(x)` has the same effect as the conversion `*reinterpret_cast<T*>(&x)` with the built-in `&` and `*` operators (and similarly for `reinterpret_cast<T&&>(x)`). — *end note*] No temporary is created, no copy is made, and constructors (12.30) or conversion functions (12.659) are not called.[73]

5.1871.1 Const cast [expr.const.cast]

1

The result of the expression `const_cast<T>(v)` is of type `T`. If `T` is an lvalue reference to object type, the result is an lvalue; if `T` is an rvalue reference to object type, the result is an xvalue; otherwise, the result is a prvalue and the lvalue-to-rvalue (4.127), array-to-pointer (4.180), and function-to-pointer (4.220) standard conversions are performed on the expression `v`. Conversions that can be performed explicitly using `const_cast` are listed below. No other conversion shall be performed explicitly using `const_cast`.

1 [*Note:* Subject to the restrictions in this section, an expression may be cast to its own type using a `const_cast` operator. — *end note*]

2

For two pointer types `T1` and `T2` where

T1 is $cv_{1,0}$ pointer to $cv_{1,1}$ pointer to $\cdots$ $cv_{1,n-1}$ pointer to $cv_{1,n}$ T

and

T2 is $cv_{2,0}$ pointer to $cv_{2,1}$ pointer to $\cdots$ $cv_{2,n-1}$ pointer to $cv_{2,n}$ T

where `T` is any object type or the `void` type and where $cv_{1,k}$ and $cv_{2,k}$ may be different cv-qualifications, a prvalue of type `T1` may be explicitly converted to the type `T2` using a `const_cast`. The result of a pointer `const_cast` refers to the original object.

1 For two object types `T1` and `T2`, if a pointer to `T1` can be explicitly converted to the type "pointer to `T2`" using a `const_cast`, then the following conversions can also be made:

(1.1) — an lvalue of type `T1` can be explicitly converted to an lvalue of type `T2` using the cast `const_cast<T2&>`;

(1.2) — a glvalue of type `T1` can be explicitly converted to an xvalue of type `T2` using the cast `const_cast<T2&&>`; and

(1.3) — if `T1` is a class type, a prvalue of type `T1` can be explicitly converted to an xvalue of type `T2` using the cast `const_cast<T2&&>`.

The result of a reference `const_cast` refers to the original object.

2 For a `const_cast` involving pointers to data members, multi-level pointers to data members and multi-level mixed pointers and pointers to data members (4.246), the rules for `const_cast` are the same as those used for pointers; the "member" aspect of a pointer to member is ignored when determining where the cv-qualifiers are added or removed by the `const_cast`. The result of a pointer to data member `const_cast` refers to the same member as the original (uncast) pointer to data member.

3 A null pointer value (4.579) is converted to the null pointer value of the destination type. The null member pointer value (4.710) is converted to the null member pointer value of the destination type.

4

73) This is sometimes referred to as a *type pun*.

[*Note:* Depending on the type of the object, a write operation through the pointer, lvalue or pointer to data member resulting from a `const_cast` that casts away a const-qualifier[74] may produce undefined behavior (7.567.0.1). — *end note*]

1

The following rules define the process known as *casting away constness*. In these rules `T`n and `X`n represent types. For two pointer types:

`X1` is `T1`$cv_{1,1}$ `*` $\cdots$ $cv_{1,N}$ `*` where `T1` is not a pointer type

`X2` is `T2`$cv_{2,1}$ `*` $\cdots$ $cv_{2,M}$ `*` where `T2` is not a pointer type

K is $\min(N, M)$

casting from `X1` to `X2` casts away constness if, for a non-pointer type `T` there does not exist an implicit conversion (Clause 4) from:

`T`$cv_{1,(N-K+1)}$ `*` $cv_{1,(N-K+2)}$ `*` $\cdots$ $cv_{1,N}$ `*`

to

`T`$cv_{2,(M-K+1)}$ `*` $cv_{2,(M-K+2)}$ `*` $\cdots$ $cv_{2,M}$ `*`

1 Casting from an lvalue of type `T1` to an lvalue of type `T2` using an lvalue reference cast or casting from an expression of type `T1` to an xvalue of type `T2` using an rvalue reference cast casts away constness if a cast from a prvalue of type "pointer to `T1`" to the type "pointer to `T2`" casts away constness.

2 Casting from a prvalue of type "pointer to data member of `X` of type `T1`" to the type "pointer to data member of `Y` of type `T2`" casts away constness if a cast from a prvalue of type "pointer to `T1`" to the type "pointer to `T2`" casts away constness.

3 For multi-level pointer to members and multi-level mixed pointers and pointer to members (4.246), the "member" aspect of a pointer to member level is ignored when determining if a `const` cv-qualifier has been cast away.

4

[*Note:* some conversions which involve only changes in cv-qualification cannot be done using `const_cast`. For instance, conversions between pointers to functions are not covered because such conversions lead to values whose use causes undefined behavior. For the same reasons, conversions between pointers to member functions, and in particular, the conversion from a pointer to a const member function to a pointer to a non-const member function, are not covered. — *end note*]

5.1897 Unary expressions [expr.unary]

1

Expressions with unary operators group right-to-left.

unary-expression:
- *postfix-expression*
- `++` *cast-expression*
- `--` *cast-expression*
- *unary-operator cast-expression*
- `sizeof` *unary-expression*
- `sizeof (` *type-id* `)`
- `sizeof ... (` *identifier* `)`
- `alignof (` *type-id* `)`
- *noexcept-expression*
- *new-expression*
- *delete-expression*

74) `const_cast` is not limited to conversions that cast away a const-qualifier.

unary-operator: one of
`* & + - ! ~`

5.1898.1 Unary operators [expr.unary.op]

1

The unary `*` operator performs *indirection*: the expression to which it is applied shall be a pointer to an object type, or a pointer to a function type and the result is an lvalue referring to the object or function to which the expression points. If the type of the expression is "pointer to `T`," the type of the result is "`T`." [*Note:* indirection through a pointer to an incomplete type (other than *cv* `void`) is valid. The lvalue thus obtained can be used in limited ways (to initialize a reference, for example); this lvalue must not be converted to a prvalue, see 4.127. — *end note*]

1

The result of each of the following unary operators is a prvalue.

1

The result of the unary `&` operator is a pointer to its operand. The operand shall be an lvalue or a *qualified-id*. If the operand is a *qualified-id* naming a non-static member `m` of some class `C` with type `T`, the result has type "pointer to member of class `C` of type `T`" and is a prvalue designating `C::m`. Otherwise, if the type of the expression is `T`, the result has type "pointer to `T`" and is a prvalue that is the address of the designated object (1.277) or a pointer to the designated function. [*Note:* In particular, the address of an object of type "*cv* `T`" is "pointer to *cv* `T`", with the same cv-qualification. — *end note*] [*Example:*

```
struct A { int i; };
struct B : A { };
... &B::i ...        // has type int A::*
```

— *end example*] [*Note:* a pointer to member formed from a `mutable` non-static data member (7.49.1) does not reflect the `mutable` specifier associated with the non-static data member. — *end note*]

1

A pointer to member is only formed when an explicit `&` is used and its operand is a *qualified-id* not enclosed in parentheses. [*Note:* that is, the expression `&(qualified-id)`, where the *qualified-id* is enclosed in parentheses, does not form an expression of type "pointer to member." Neither does `qualified-id`, because there is no implicit conversion from a *qualified-id* for a non-static member function to the type "pointer to member function" as there is from an lvalue of function type to the type "pointer to function" (4.220). Nor is `&unqualified-id` a pointer to member, even within the scope of the *unqualified-id*'s class. — *end note*]

1

If `&` is applied to an lvalue of incomplete class type and the complete type declares `operator&()`, it is unspecified whether the operator has the built-in meaning or the operator function is called. The operand of `&` shall not be a bit-field.

1

The address of an overloaded function (Clause 13) can be taken only in a context that uniquely determines which version of the overloaded function is referred to (see 13.1084). [*Note:* since the context might determine whether the operand is a static or non-static member function, the context can also affect whether the expression has type "pointer to function" or "pointer to member function." — *end note*]

1

The operand of the unary `+` operator shall have arithmetic, unscoped enumeration, or pointer type and the result is the value of the argument. Integral promotion is performed on integral or enumeration operands. The type of the result is the type of the promoted operand.

1

The operand of the unary `-` operator shall have arithmetic or unscoped enumeration type and the result is the negation of its operand. Integral promotion is performed on integral or enumeration operands. The negative of an unsigned quantity is computed by subtracting its value from 2^n, where n is the number of bits in the promoted operand. The type of the result is the type of the promoted operand.

1

The operand of the logical negation operator `!` is contextually converted to `bool` (Clause 4); its value is `true` if the converted operand is `false` and `false` otherwise. The type of the result is `bool`.

1

The operand of `~` shall have integral or unscoped enumeration type; the result is the one's complement of its operand. Integral promotions are performed. The type of the result is the type of the promoted operand. There is an ambiguity in the *unary-expression* `~X()`, where `X` is a *class-name* or *decltype-specifier*. The ambiguity is resolved in favor of treating `~` as a unary complement rather than treating `~X` as referring to a destructor.

5.2054.1 Increment and decrement [expr.pre.incr]

1

The operand of prefix `++` is modified by adding `1`, or set to `true` if it is `bool` (this use is deprecated). The operand shall be a modifiable lvalue. The type of the operand shall be an arithmetic type or a pointer to a completely-defined object type. The result is the updated operand; it is an lvalue, and it is a bit-field if the operand is a bit-field. If `x` is not of type `bool`, the expression `++x` is equivalent to `x+=1` [*Note:* See the discussions of addition (5.2882) and assignment operators (5.3483) for information on conversions. — *end note*]

1

The operand of prefix `--` is modified by subtracting `1`. The operand shall not be of type `bool`. The requirements on the operand of prefix `--` and the properties of its result are otherwise the same as those of prefix `++`. [*Note:* For postfix increment and decrement, see 5.1264.1. — *end note*]

5.2075.1 Sizeof [expr.sizeof]

1

The `sizeof` operator yields the number of bytes in the object representation of its operand. The operand is either an expression, which is an unevaluated operand (Clause 5), or a parenthesized *type-id*. The `sizeof` operator shall not be applied to an expression that has function or incomplete type, to an enumeration type whose underlying type is not fixed before all its enumerators have been declared, to the parenthesized name of such types, or to a glvalue that designates a bit-field. `sizeof(char)`, `sizeof(signed char)` and `sizeof(unsigned char)` are `1`. The result of `sizeof` applied to any other fundamental type (3.3151.1) is implementation-defined. [*Note:* in particular, `sizeof(bool)`, `sizeof(char16_t)`, `sizeof(char32_t)`, and `sizeof(wchar_t)` are implementation-defined.[75] — *end note*] [*Note:* See 1.277 for the definition of *byte* and 3.2955 for the definition of *object representation*. — *end note*]

1

When applied to a reference or a reference type, the result is the size of the referenced type. When applied to a class, the result is the number of bytes in an object of that class including any padding required for placing objects of that type in an array. The size of a most derived class shall be greater than zero (1.319).

75) `sizeof(bool)` is not required to be `1`.

The result of applying `sizeof` to a base class subobject is the size of the base class type.[76] When applied to an array, the result is the total number of bytes in the array. This implies that the size of an array of n elements is n times the size of an element.

1

The `sizeof` operator can be applied to a pointer to a function, but shall not be applied directly to a function.

1

The lvalue-to-rvalue (4.127), array-to-pointer (4.180), and function-to-pointer (4.220) standard conversions are not applied to the operand of `sizeof`.

1

The identifier in a `sizeof...` expression shall name a parameter pack. The `sizeof...` operator yields the number of arguments provided for the parameter pack *identifier*. A `sizeof...` expression is a pack expansion (14.703.1). [*Example:*

```
template<class... Types>
struct count {
  static const std::size_t value = sizeof...(Types);
};
```

— *end example*]

1

The result of `sizeof` and `sizeof...` is a constant of type `std::size_t`. [*Note:* `std::size_t` is defined in the standard header `<cstddef>` (18.2). — *end note*]

5.2221.1 New [expr.new]

1

The *new-expression* attempts to create an object of the *type-id* (8.32) or *new-type-id* to which it is applied. The type of that object is the *allocated type*. This type shall be a complete object type, but not an abstract class type or array thereof (1.319, 3.2955, 10.276). It is implementation-defined whether over-aligned types are supported (3.3924). [*Note:* because references are not objects, references cannot be created by *new-expressions*. — *end note*] [*Note:* the *type-id* may be a cv-qualified type, in which case the object created by the *new-expression* has a cv-qualified type. — *end note*]

new-expression:
  `::`$_{opt}$`new` *new-placement*$_{opt}$ *new-type-id* *new-initializer*$_{opt}$
  `::`$_{opt}$`new` *new-placement*$_{opt}$`(` *type-id* `)` *new-initializer*$_{opt}$

new-placement:
  `(` *expression-list* `)`

new-type-id:
  type-specifier-seq *new-declarator*$_{opt}$

new-declarator:
  ptr-operator *new-declarator*$_{opt}$
  noptr-new-declarator

noptr-new-declarator:
  `[` *expression* `]` *attribute-specifier-seq*$_{opt}$
  noptr-new-declarator `[` *constant-expression* `]` *attribute-specifier-seq*$_{opt}$

new-initializer:
  `(` *expression-list*$_{opt}$`)`
  braced-init-list

76) The actual size of a base class subobject may be less than the result of applying `sizeof` to the subobject, due to virtual base classes and less strict padding requirements on base class subobjects.

Entities created by a *new-expression* have dynamic storage duration (3.2331.1). [*Note:* the lifetime of such an entity is not necessarily restricted to the scope in which it is created. — *end note*] If the entity is a non-array object, the *new-expression* returns a pointer to the object created. If it is an array, the *new-expression* returns a pointer to the initial element of the array.

1

If the `auto` *type-specifier* appears in the *type-specifier-seq* of a *new-type-id* or *type-id* of a *new-expression*, the *new-expression* shall contain a *new-initializer* of the form

`(` *assignment-expression* `)`

The allocated type is deduced from the *new-initializer* as follows: Let `e` be the *assignment-expression* in the *new-initializer* and `T` be the *new-type-id* or *type-id* of the *new-expression*, then the allocated type is the type deduced for the variable `x` in the invented declaration (7.832.0.1):

```
T x(e);
```

[*Example:*

```
new auto(1);                    // allocated type is int
auto x = new auto('a');         // allocated type is char, x is of type char*
```

— *end example*]

1

The *new-type-id* in a *new-expression* is the longest possible sequence of *new-declarator*s. [*Note:* this prevents ambiguities between the declarator operators `&`, `&&`, `*`, and `[]` and their expression counterparts. — *end note*] [*Example:*

```
new int * i;                    // syntax error: parsed as (new int*) i, not as (new int)*i
```

The `*` is the pointer declarator and not the multiplication operator. — *end example*]

1

[*Note:* parentheses in a *new-type-id* of a *new-expression* can have surprising effects. [*Example:*

```
new int(*[10])();               // error
```

is ill-formed because the binding is

```
(new int) (*[10])();            // error
```

Instead, the explicitly parenthesized version of the `new` operator can be used to create objects of compound types (3.3568.1):

```
new (int (*[10])());
```

allocates an array of `10` pointers to functions (taking no argument and returning `int`. — *end example*] — *end note*]

1

When the allocated object is an array (that is, the *noptr-new-declarator* syntax is used or the *new-type-id* or *type-id* denotes an array type), the *new-expression* yields a pointer to the initial element (if any) of the array. [*Note:* both `new int` and `new int[10]` have type `int*` and the type of `new int[i][10]` is `int (*)[10]` — *end note*] The *attribute-specifier-seq* in a *noptr-new-declarator* appertains to the associated array type.

1

Every *constant-expression* in a *noptr-new-declarator* shall be a converted constant expression (5.3643) of type `std::size_t` and shall evaluate to a strictly positive value. The *expression* in a *noptr-new-declarator*

is implicitly converted to `std::size_t`. [*Example:* given the definition `int n = 42`, `new float[n][5]` is well-formed (because `n` is the *expression* of a *noptr-new-declarator*), but `new float[5][n]` is ill-formed (because `n` is not a constant expression). — *end example*]

1

The *expression* in a *noptr-new-declarator* is erroneous if:

— the expression is of non-class type and its value before converting to `std::size_t` is less than zero;

— the expression is of class type and its value before application of the second standard conversion (13.844.0.0.1)[77] is less than zero;

— its value is such that the size of the allocated object would exceed the implementation-defined limit (annex B); or

— the *new-initializer* is a *braced-init-list* and the number of array elements for which initializers are provided (including the terminating `'\0'` in a string literal (2.453.1)) exceeds the number of elements to initialize.

If the *expression*, after converting to `std::size_t`, is a core constant expression and the expression is erroneous, the program is ill-formed. Otherwise, a *new-expression* with an erroneous expression does not call an allocation function and terminates by throwing an exception of a type that would match a handler (15.146) of type `std::bad_array_new_length` (18.225.0.1). When the value of the *expression* is zero, the allocation function is called to allocate an array with no elements.

1

A *new-expression* may obtain storage for the object by calling an *allocation function* (3.2361.0.1). If the *new-expression* terminates by throwing an exception, it may release storage by calling a deallocation function (3.2428.0.1). If the allocated type is a non-array type, the allocation function's name is `operator new` and the deallocation function's name is `operator delete`. If the allocated type is an array type, the allocation function's name is `operator new[]` and the deallocation function's name is `operator delete[]`. [*Note:* an implementation shall provide default definitions for the global allocation functions (3.2331.1, 18.159.1.1, 18.196.0.1). A C++ program can provide alternative definitions of these functions (17.428.0.3) and/or class-specific versions (12.988). — *end note*]

1

If the *new-expression* begins with a unary `::` operator, the allocation function's name is looked up in the global scope. Otherwise, if the allocated type is a class type `T` or array thereof, the allocation function's name is looked up in the scope of `T`. If this lookup fails to find the name, or if the allocated type is not a class type, the allocation function's name is looked up in the global scope.

1

An implementation is allowed to omit a call to a replaceable global allocation function (18.159.1.1, 18.196.0.1). When it does so, the storage is instead provided by the implementation or provided by extending the allocation of another *new-expression*. The implementation may extend the allocation of a *new-expression* `e1` to provide storage for a *new-expression* `e2` if the following would be true were the allocation not extended:

— the evaluation of `e1` is sequenced before the evaluation of `e2`, and

— `e2` is evaluated whenever `e1` obtains storage, and

— both `e1` and `e2` invoke the same replaceable global allocation function, and

77) If the conversion function returns a signed integer type, the second standard conversion converts to the unsigned type `std::size_t` and thus thwarts any attempt to detect a negative value afterwards.

— if the allocation function invoked by `e1` and `e2` is throwing, any exceptions thrown in the evaluation of either `e1` or `e2` would be first caught in the same handler, and

— the pointer values produced by `e1` and `e2` are operands to evaluated *delete-expression*s, and

— the evaluation of `e2` is sequenced before the evaluation of the *delete-expression* whose operand is the pointer value produced by `e1`.

[*Example:*

```
void mergeable(int x) {
  // These allocations are safe for merging:
  std::unique_ptr<char[]> a{new (std::nothrow) char[8]};
  std::unique_ptr<char[]> b{new (std::nothrow) char[8]};
  std::unique_ptr<char[]> c{new (std::nothrow) char[x]};

  g(a.get(), b.get(), c.get());
}

void unmergeable(int x) {
  std::unique_ptr<char[]> a{new char[8]};
  try {
    // Merging this allocation would change its catch handler.
    std::unique_ptr<char[]> b{new char[x]};
  } catch (const std::bad_alloc& e) {
    std::cerr << "Allocation failed: " << e.what() << std::endl;
    throw;
  }
}
```

— *end example*]

1

When a *new-expression* calls an allocation function and that allocation has not been extended, the *new-expression* passes the amount of space requested to the allocation function as the first argument of type `std::size_t`. That argument shall be no less than the size of the object being created; it may be greater than the size of the object being created only if the object is an array. For arrays of `char` and `unsigned char`, the difference between the result of the *new-expression* and the address returned by the allocation function shall be an integral multiple of the strictest fundamental alignment requirement (3.3924) of any object type whose size is no greater than the size of the array being created. [*Note:* Because allocation functions are assumed to return pointers to storage that is appropriately aligned for objects of any type with fundamental alignment, this constraint on array allocation overhead permits the common idiom of allocating character arrays into which objects of other types will later be placed. — *end note*]

1

When a *new-expression* calls an allocation function and that allocation has been extended, the size argument to the allocation call shall be no greater than the sum of the sizes for the omitted calls as specified above, plus the size for the extended call had it not been extended, plus any padding necessary to align the allocated objects within the allocated memory.

1

The *new-placement* syntax is used to supply additional arguments to an allocation function. If used, overload resolution is performed on a function call created by assembling an argument list consisting of the amount of space requested (the first argument) and the expressions in the *new-placement* part of the *new-expression* (the second and succeeding arguments). The first of these arguments has type `std::size_t` and the remaining arguments have the corresponding types of the expressions in the *new-placement*.

1

[*Example:*

— `new T` results in a call of `operator new(sizeof(T))`,

— `new(2,f) T` results in a call of `operator new(sizeof(T),2,f)`,

— `new T[5]` results in a call of `operator new[](sizeof(T)*5+x)`, and

— `new(2,f) T[5]` results in a call of `operator new[](sizeof(T)*5+y,2,f)`.

Here, `x` and `y` are non-negative unspecified values representing array allocation overhead; the result of the *new-expression* will be offset by this amount from the value returned by `operator new[]`. This overhead may be applied in all array *new-expressions*, including those referencing the library function `operator new[](std::size_t, void*)` and other placement allocation functions. The amount of overhead may vary from one invocation of `new` to another. — *end example*]

1

[*Note:* unless an allocation function is declared with a non-throwing *exception-specification* (15.228), it indicates failure to allocate storage by throwing a `std::bad_alloc` exception (Clause 15, 18.223.1.1); it returns a non-null pointer otherwise. If the allocation function is declared with a non-throwing *exception-specification*, it returns null to indicate failure to allocate storage and a non-null pointer otherwise. — *end note*] If the allocation function returns null, initialization shall not be done, the deallocation function shall not be called, and the value of the *new-expression* shall be null.

1

[*Note:* when the allocation function returns a value other than null, it must be a pointer to a block of storage in which space for the object has been reserved. The block of storage is assumed to be appropriately aligned and of the requested size. The address of the created object will not necessarily be the same as that of the block if the object is an array. — *end note*]

1

A *new-expression* that creates an object of type `T` initializes that object as follows:

— If the *new-initializer* is omitted, the object is default-initialized (8.821). [*Note:* If no initialization is performed, the object has an indeterminate value. — *end note*]

— Otherwise, the *new-initializer* is interpreted according to the initialization rules of 8.821 for direct-initialization.

1

The invocation of the allocation function is indeterminately sequenced with respect to the evaluations of expressions in the *new-initializer*. Initialization of the allocated object is sequenced before the value computation of the *new-expression*. It is unspecified whether expressions in the *new-initializer* are evaluated if the allocation function returns the null pointer or exits using an exception.

1

If the *new-expression* creates an object or an array of objects of class type, access and ambiguity control are done for the allocation function, the deallocation function (12.988), and the constructor (12.30). If the *new-expression* creates an array of objects of class type, the destructor is potentially invoked (12.807).

1

If any part of the object initialization described above[78] terminates by throwing an exception, storage has been obtained for the object, and a suitable deallocation function can be found, the deallocation function is

78) This may include evaluating a *new-initializer* and/or calling a constructor.

called to free the memory in which the object was being constructed, after which the exception continues to propagate in the context of the *new-expression*. If no unambiguous matching deallocation function can be found, propagating the exception does not cause the object's memory to be freed. [*Note:* This is appropriate when the called allocation function does not allocate memory; otherwise, it is likely to result in a memory leak. — *end note*]

1 If the *new-expression* begins with a unary `::` operator, the deallocation function's name is looked up in the global scope. Otherwise, if the allocated type is a class type `T` or an array thereof, the deallocation function's name is looked up in the scope of `T`. If this lookup fails to find the name, or if the allocated type is not a class type or array thereof, the deallocation function's name is looked up in the global scope.

2

A declaration of a placement deallocation function matches the declaration of a placement allocation function if it has the same number of parameters and, after parameter transformations (8.360.1), all parameter types except the first are identical. If the lookup finds a single matching deallocation function, that function will be called; otherwise, no deallocation function will be called. If the lookup finds the two-parameter form of a usual deallocation function (3.2428.0.1) and that function, considered as a placement deallocation function, would have been selected as a match for the allocation function, the program is ill-formed. For a non-placement allocation function, the normal deallocation function lookup is used to find the matching deallocation function (5.2455.1) [*Example:*

```
struct S {
  // Placement allocation function:
  static void* operator new(std::size_t, std::size_t);

  // Usual (non-placement) deallocation function:
  static void operator delete(void*, std::size_t);
};

S* p = new (0) S;    // ill-formed: non-placement deallocation function matches
                     // placement allocation function
```

— *end example*]

1

If a *new-expression* calls a deallocation function, it passes the value returned from the allocation function call as the first argument of type `void*`. If a placement deallocation function is called, it is passed the same additional arguments as were passed to the placement allocation function, that is, the same arguments as those specified with the *new-placement* syntax. If the implementation is allowed to make a copy of any argument as part of the call to the allocation function, it is allowed to make a copy (of the same original value) as part of the call to the deallocation function or to reuse the copy made as part of the call to the allocation function. If the copy is elided in one place, it need not be elided in the other.

5.2455.1 Delete [expr.delete]

1

The *delete-expression* operator destroys a most derived object (1.319) or array created by a *new-expression*.

delete-expression:
  `::`$_{opt}$`delete` *cast-expression*
  `::`$_{opt}$`delete [ ]` *cast-expression*

The first alternative is for non-array objects, and the second is for arrays. Whenever the `delete` keyword is immediately followed by empty square brackets, it shall be interpreted as the second alternative.[79] The

79) A lambda expression with a *lambda-introducer* that consists of empty square brackets can follow the `delete` keyword if

operand shall be of pointer to object type or of class type. If of class type, the operand is contextually implicitly converted (Clause 4) to a pointer to object type.[80] The *delete-expression*'s result has type `void`.

1

If the operand has a class type, the operand is converted to a pointer type by calling the above-mentioned conversion function, and the converted operand is used in place of the original operand for the remainder of this section. In the first alternative (*delete object*), the value of the operand of `delete` may be a null pointer value, a pointer to a non-array object created by a previous *new-expression*, or a pointer to a subobject (1.319) representing a base class of such an object (Clause 10). If not, the behavior is undefined. In the second alternative (*delete array*), the value of the operand of `delete` may be a null pointer value or a pointer value that resulted from a previous array *new-expression*.[81] If not, the behavior is undefined. [*Note:* this means that the syntax of the *delete-expression* must match the type of the object allocated by `new`, not the syntax of the *new-expression*. — *end note*] [*Note:* a pointer to a `const` type can be the operand of a *delete-expression*; it is not necessary to cast away the constness (5.1871.1) of the pointer expression before it is used as the operand of the *delete-expression*. — *end note*]

1

In the first alternative (*delete object*), if the static type of the object to be deleted is different from its dynamic type, the static type shall be a base class of the dynamic type of the object to be deleted and the static type shall have a virtual destructor or the behavior is undefined. In the second alternative (*delete array*) if the dynamic type of the object to be deleted differs from its static type, the behavior is undefined.

1

The *cast-expression* in a *delete-expression* shall be evaluated exactly once.

1

If the object being deleted has incomplete class type at the point of deletion and the complete class has a non-trivial destructor or a deallocation function, the behavior is undefined.

1

If the value of the operand of the *delete-expression* is not a null pointer value, the *delete-expression* will invoke the destructor (if any) for the object or the elements of the array being deleted. In the case of an array, the elements will be destroyed in order of decreasing address (that is, in reverse order of the completion of their constructor; see 12.1026.1).

1

If the value of the operand of the *delete-expression* is not a null pointer value, then:

— If the allocation call for the *new-expression* for the object to be deleted was not omitted and the allocation was not extended (5.2221.1), the *delete-expression* shall call a deallocation function (3.2428.0.1). The value returned from the allocation call of the *new-expression* shall be passed as the first argument to the deallocation function.

— Otherwise, if the allocation was extended or was provided by extending the allocation of another *new-expression*, and the *delete-expression* for every other pointer value produced by a *new-expression* that had storage provided by the extended *new-expression* has been evaluated, the *delete-expression* shall call a deallocation function. The value returned from the allocation call of the extended *new-expression* shall be passed as the first argument to the deallocation function.

— Otherwise, the *delete-expression* will not call a *deallocation function* (3.2428.0.1).

---

the lambda expression is enclosed in parentheses.

80) This implies that an object cannot be deleted using a pointer of type `void*` because `void` is not an object type.

81) For non-zero-length arrays, this is the same as a pointer to the first element of the array created by that *new-expression*. Zero-length arrays do not have a first element.

Otherwise, it is unspecified whether the deallocation function will be called. [*Note:* The deallocation function is called regardless of whether the destructor for the object or some element of the array throws an exception. — *end note*]

1 [*Note:* An implementation provides default definitions of the global deallocation functions `operator delete()` for non-arrays (18.159.1.1) and `operator delete[]()` for arrays (18.196.0.1). A C++ program can provide alternative definitions of these functions (17.428.0.3), and/or class-specific versions (12.988). — *end note*]

2

When the keyword `delete` in a *delete-expression* is preceded by the unary `::` operator, the deallocation function's name is looked up in global scope. Otherwise, the lookup considers class-specific deallocation functions (12.988). If no class-specific deallocation function is found, the deallocation function's name is looked up in global scope.

1

If the type is complete and if deallocation function lookup finds both a usual deallocation function with only a pointer parameter and a usual deallocation function with both a pointer parameter and a size parameter, then the selected deallocation function shall be the one with two parameters. Otherwise, the selected deallocation function shall be the function with one parameter.

1

When a *delete-expression* is executed, the selected deallocation function shall be called with the address of the block of storage to be reclaimed as its first argument and (if the two-parameter deallocation function is used) the size of the block as its second argument.[82]

1

Access and ambiguity control are done for both the deallocation function and the destructor (12.807, 12.988).

5.2747.1 Alignof [expr.alignof]

1

An `alignof` expression yields the alignment requirement of its operand type. The operand shall be a *type-id* representing a complete object type, or an array thereof, or a reference to one of those types.

1 The result is an integral constant of type `std::size_t`.

2

When `alignof` is applied to a reference type, the result is the alignment of the referenced type. When `alignof` is applied to an array type, the result is the alignment of the element type.

5.2750.1 `noexcept` operator [expr.unary.noexcept]

1

The `noexcept` operator determines whether the evaluation of its operand, which is an unevaluated operand (Clause 5), can throw an exception (15.7).

noexcept-expression:
`noexcept (` *expression* `)`

1

The result of the `noexcept` operator is a constant of type `bool` and is a prvalue.

1

82) If the static type of the object to be deleted is complete and is different from the dynamic type, and the destructor is not virtual, the size might be incorrect, but that case is already undefined, as stated above.

The result of the `noexcept` operator is `false` if in a potentially-evaluated context the *expression* would contain

— a potentially-evaluated call[83] to a function, member function, function pointer, or member function pointer that does not have a non-throwing *exception-specification* (15.228), unless the call is a constant expression (5.3643),

— a potentially-evaluated *throw-expression* (15.7),

— a potentially-evaluated `dynamic_cast` expression `dynamic_cast<T>(v)`, where `T` is a reference type, that requires a run-time check (5.1295.1), or

— a potentially-evaluated `typeid` expression (5.1393.1) applied to a glvalue expression whose type is a polymorphic class type (10.113).

Otherwise, the result is `true`.

5.2765 Explicit type conversion (cast notation) [expr.cast]

1

The result of the expression `(T)` *cast-expression* is of type `T`. The result is an lvalue if `T` is an lvalue reference type or an rvalue reference to function type and an xvalue if `T` is an rvalue reference to object type; otherwise the result is a prvalue. [*Note:* if `T` is a non-class type that is cv-qualified, the *cv-qualifiers* are discarded when determining the type of the resulting prvalue; see Clause 5. — *end note*]

1

An explicit type conversion can be expressed using functional notation (5.1108.1), a type conversion operator (`dynamic_cast`, `static_cast`, `reinterpret_cast`, `const_cast`), or the *cast* notation.

cast-expression:
  unary-expression
  `(` *type-id* `)` *cast-expression*

1

Any type conversion not mentioned below and not explicitly defined by the user (12.659) is ill-formed.

1

The conversions performed by

— a `const_cast` (5.1871.1),

— a `static_cast` (5.1443.1),

— a `static_cast` followed by a `const_cast`,

— a `reinterpret_cast` (5.1644.1), or

— a `reinterpret_cast` followed by a `const_cast`,

can be performed using the cast notation of explicit type conversion. The same semantic restrictions and behaviors apply, with the exception that in performing a `static_cast` in the following situations the conversion is valid even if the base class is inaccessible:

— a pointer to an object of derived class type or an lvalue or rvalue of derived class type may be explicitly converted to a pointer or reference to an unambiguous base class type, respectively;

— a pointer to member of derived class type may be explicitly converted to a pointer to member of an unambiguous non-virtual base class type;

83) This includes implicit calls such as the call to an allocation function in a *new-expression*.

— a pointer to an object of an unambiguous non-virtual base class type, a glvalue of an unambiguous non-virtual base class type, or a pointer to member of an unambiguous non-virtual base class type may be explicitly converted to a pointer, a reference, or a pointer to member of a derived class type, respectively.

If a conversion can be interpreted in more than one of the ways listed above, the interpretation that appears first in the list is used, even if a cast resulting from that interpretation is ill-formed. If a conversion can be interpreted in more than one way as a `static_cast` followed by a `const_cast`, the conversion is ill-formed. [*Example:*

```
struct A { };
struct I1 : A { };
struct I2 : A { };
struct D : I1, I2 { };
A* foo( D* p ) {
  return (A*)( p ); // ill-formed static_cast interpretation
}
```

— *end example*]

1

The operand of a cast using the cast notation can be a prvalue of type "pointer to incomplete class type". The destination type of a cast using the cast notation can be "pointer to incomplete class type". If both the operand and destination types are class types and one or both are incomplete, it is unspecified whether the `static_cast` or the `reinterpret_cast` interpretation is used, even if there is an inheritance relationship between the two classes. [*Note:* For example, if the classes were defined later in the translation unit, a multi-pass compiler would be permitted to interpret a cast between pointers to the classes as if the class types were complete at the point of the cast. — *end note*]

5.2832 Pointer-to-member operators [expr.mptr.oper]

1

The pointer-to-member operators `->*` and `.*` group left-to-right.

pm-expression:
  cast-expression
  pm-expression `.*` *cast-expression*
  pm-expression `->*` *cast-expression*

1

The binary operator `.*` binds its second operand, which shall be of type "pointer to member of `T`" to its first operand, which shall be of class `T` or of a class of which `T` is an unambiguous and accessible base class. The result is an object or a function of the type specified by the second operand.

1

The binary operator `->*` binds its second operand, which shall be of type "pointer to member of `T`" to its first operand, which shall be of type "pointer to `T`" or "pointer to a class of which `T` is an unambiguous and accessible base class." The expression `E1->*E2` is converted into the equivalent form `(*(E1)).*E2`.

1

Abbreviating *pm-expression*`.*`*cast-expression* as `E1.*E2`, `E1` is called the *object expression*. If the dynamic type of `E1` does not contain the member to which `E2` refers, the behavior is undefined.

1

The restrictions on *cv*-qualification, and the manner in which the *cv*-qualifiers of the operands are combined to produce the *cv*-qualifiers of the result, are the same as the rules for `E1.E2` given in 5.1194.1. [*Note:* it is

not possible to use a pointer to member that refers to a `mutable` member to modify a `const` class object. For example,

```
struct S {
  S() : i(0) { }
  mutable int i;
};
void f()
{
const S cs;
int S::* pm = &S::i;            // pm refers to mutable member S::i
cs.*pm = 88;                    // ill-formed: cs is a const object
}
```

— *end note*]

1

If the result of `.*` or `->*` is a function, then that result can be used only as the operand for the function call operator `()`. [*Example:*

```
(ptr_to_obj->*ptr_to_mfct)(10);
```

calls the member function denoted by `ptr_to_mfct` for the object pointed to by `ptr_to_obj`. — *end example*] In a `.*` expression whose object expression is an rvalue, the program is ill-formed if the second operand is a pointer to member function with *ref-qualifier* `&`. In a `.*` expression whose object expression is an lvalue, the program is ill-formed if the second operand is a pointer to member function with *ref-qualifier* `&&`. The result of a `.*` expression whose second operand is a pointer to a data member is an lvalue if the first operand is an lvalue and an xvalue otherwise. The result of a `.*` expression whose second operand is a pointer to a member function is a prvalue. If the second operand is the null pointer to member value (4.710), the behavior is undefined.

5.2851 Multiplicative operators [expr.mul]

1 The multiplicative operators `*`, `/`, and `%` group left-to-right.

multiplicative-expression:
- *pm-expression*
- *multiplicative-expression* `*` *pm-expression*
- *multiplicative-expression* `/` *pm-expression*
- *multiplicative-expression* `%` *pm-expression*

2

The operands of `*` and `/` shall have arithmetic or unscoped enumeration type; the operands of `%` shall have integral or unscoped enumeration type. The usual arithmetic conversions are performed on the operands and determine the type of the result.

1 The binary `*` operator indicates multiplication.

2

The binary `/` operator yields the quotient, and the binary `%` operator yields the remainder from the division of the first expression by the second. If the second operand of `/` or `%` is zero the behavior is undefined. For integral operands the `/` operator yields the algebraic quotient with any fractional part discarded;[84] if the quotient `a/b` is representable in the type of the result, `(a/b)*b + a%b` is equal to `a`; otherwise, the behavior of both `a/b` and `a%b` is undefined.

84) This is often called truncation towards zero.

5.2882 Additive operators [expr.add]

1

The additive operators + and - group left-to-right. The usual arithmetic conversions are performed for operands of arithmetic or enumeration type.

additive-expression:
multiplicative-expression
additive-expression + *multiplicative-expression*
additive-expression - *multiplicative-expression*

For addition, either both operands shall have arithmetic or unscoped enumeration type, or one operand shall be a pointer to a completely-defined object type and the other shall have integral or unscoped enumeration type.

1

For subtraction, one of the following shall hold:

— both operands have arithmetic or unscoped enumeration type; or

— both operands are pointers to cv-qualified or cv-unqualified versions of the same completely-defined object type; or

— the left operand is a pointer to a completely-defined object type and the right operand has integral or unscoped enumeration type.

1

The result of the binary + operator is the sum of the operands. The result of the binary - operator is the difference resulting from the subtraction of the second operand from the first.

1

For the purposes of these operators, a pointer to a nonarray object behaves the same as a pointer to the first element of an array of length one with the type of the object as its element type.

1

When an expression that has integral type is added to or subtracted from a pointer, the result has the type of the pointer operand. If the pointer operand points to an element of an array object, and the array is large enough, the result points to an element offset from the original element such that the difference of the subscripts of the resulting and original array elements equals the integral expression. In other words, if the expression P points to the i-th element of an array object, the expressions (P)+N (equivalently, N+(P)) and (P)-N (where N has the value n) point to, respectively, the $i + n$-th and $i - n$-th elements of the array object, provided they exist. Moreover, if the expression P points to the last element of an array object, the expression (P)+1 points one past the last element of the array object, and if the expression Q points one past the last element of an array object, the expression (Q)-1 points to the last element of the array object. If both the pointer operand and the result point to elements of the same array object, or one past the last element of the array object, the evaluation shall not produce an overflow; otherwise, the behavior is undefined.

1

When two pointers to elements of the same array object are subtracted, the result is the difference of the subscripts of the two array elements. The type of the result is an implementation-defined signed integral type; this type shall be the same type that is defined as std::ptrdiff_t in the <cstddef> header (18.2). As with any other arithmetic overflow, if the result does not fit in the space provided, the behavior is undefined. In other words, if the expressions P and Q point to, respectively, the i-th and j-th elements of an array object, the expression (P)-(Q) has the value $i - j$ provided the value fits in an object of type std::ptrdiff_t. Moreover, if the expression P points either to an element of an array object or one past the last element of

an array object, and the expression `Q` points to the last element of the same array object, the expression `((Q)+1)-(P)` has the same value as `((Q)-(P))+1` and as `-((P)-((Q)+1))`, and has the value zero if the expression `P` points one past the last element of the array object, even though the expression `(Q)+1` does not point to an element of the array object. Unless both pointers point to elements of the same array object, or one past the last element of the array object, the behavior is undefined.[85]

1

For addition or subtraction, if the expressions `P` or `Q` have type "pointer to *cv* `T`", where `T` is different from the cv-unqualified array element type, the behavior is undefined. [*Note:* In particular, a pointer to a base class cannot be used for pointer arithmetic when the array contains objects of a derived class type. — *end note*]

1

If the value 0 is added to or subtracted from a pointer value, the result compares equal to the original pointer value. If two pointers point to the same object or both point one past the end of the same array or both are null, and the two pointers are subtracted, the result compares equal to the value 0 converted to the type `std::ptrdiff_t`.

5.3085 Shift operators [expr.shift]

1

The shift operators `<<` and `>>` group left-to-right.

shift-expression:
  additive-expression
  shift-expression `<<` *additive-expression*
  shift-expression `>>` *additive-expression*

The operands shall be of integral or unscoped enumeration type and integral promotions are performed. The type of the result is that of the promoted left operand. The behavior is undefined if the right operand is negative, or greater than or equal to the length in bits of the promoted left operand.

1

The value of `E1 << E2` is `E1` left-shifted `E2` bit positions; vacated bits are zero-filled. If `E1` has an unsigned type, the value of the result is $E1 \times 2^{E2}$, reduced modulo one more than the maximum value representable in the result type. Otherwise, if `E1` has a signed type and non-negative value, and $E1 \times 2^{E2}$ is representable in the corresponding unsigned type of the result type, then that value, converted to the result type, is the resulting value; otherwise, the behavior is undefined.

1

The value of `E1 >> E2` is `E1` right-shifted `E2` bit positions. If `E1` has an unsigned type or if `E1` has a signed type and a non-negative value, the value of the result is the integral part of the quotient of $E1/2^{E2}$. If `E1` has a signed type and a negative value, the resulting value is implementation-defined.

5.3203 Relational operators [expr.rel]

1

The relational operators group left-to-right. [*Example:* `a<b<c` means `(a<b)<c` and *not* `(a<b)&&(b<c)`. — *end example*]

85) Another way to approach pointer arithmetic is first to convert the pointer(s) to character pointer(s): In this scheme the integral value of the expression added to or subtracted from the converted pointer is first multiplied by the size of the object originally pointed to, and the resulting pointer is converted back to the original type. For pointer subtraction, the result of the difference between the character pointers is similarly divided by the size of the object originally pointed to.

When viewed in this way, an implementation need only provide one extra byte (which might overlap another object in the program) just after the end of the object in order to satisfy the "one past the last element" requirements.

relational-expression:
shift-expression
relational-expression `<` *shift-expression*
relational-expression `>` *shift-expression*
relational-expression `<=` *shift-expression*
relational-expression `>=` *shift-expression*

The operands shall have arithmetic, enumeration, or pointer type. The operators `<` (less than), `>` (greater than), `<=` (less than or equal to), and `>=` (greater than or equal to) all yield `false` or `true`. The type of the result is `bool`.

1

The usual arithmetic conversions are performed on operands of arithmetic or enumeration type. If both operands are pointers, pointer conversions (4.579) and qualification conversions (4.246) are performed to bring them to their composite pointer type (Clause 5). After conversions, the operands shall have the same type.

1

Comparing pointers to objects is defined as follows:

— If two pointers point to different elements of the same array, or to subobjects thereof, the pointer to the element with the higher subscript compares greater.

— If one pointer points to an element of an array, or to a subobject thereof, and another pointer points one past the last element of the array, the latter pointer compares greater.

— If two pointers point to different non-static data members of the same object, or to subobjects of such members, recursively, the pointer to the later declared member compares greater provided the two members have the same access control (Clause 11) and provided their class is not a union.

1

If two operands `p` and `q` compare equal (5.3247), `p<=q` and `p>=q` both yield `true` and `p<q` and `p>q` both yield `false`. Otherwise, if a pointer `p` compares greater than a pointer `q`, `p>=q`, `p>q`, `q<=p`, and `q<p` all yield `true` and `p<=q`, `p<q`, `q>=p`, and `q>p` all yield `false`. Otherwise, the result of each of the operators is unspecified.

1

If both operands (after conversions) are of arithmetic or enumeration type, each of the operators shall yield `true` if the specified relationship is true and `false` if it is false.

5.3247 Equality operators [expr.eq]

equality-expression:
relational-expression
equality-expression `==` *relational-expression*
equality-expression `!=` *relational-expression*

1

The `==` (equal to) and the `!=` (not equal to) operators group left-to-right. The operands shall have arithmetic, enumeration, pointer, or pointer to member type, or type `std::nullptr_t`. The operators `==` and `!=` both yield `true` or `false`, i.e., a result of type `bool`. In each case below, the operands shall have the same type after the specified conversions have been applied.

1

If at least one of the operands is a pointer, pointer conversions (4.579) and qualification conversions (4.246) are performed on both operands to bring them to their composite pointer type (Clause 5). Comparing pointers is defined as follows: Two pointers compare equal if they are both null, both point to the same function, or both represent the same address (3.3568.1), otherwise they compare unequal.

1

If at least one of the operands is a pointer to member, pointer to member conversions (4.710) and qualification conversions (4.246) are performed on both operands to bring them to their composite pointer type (Clause 5). Comparing pointers to members is defined as follows:

— If two pointers to members are both the null member pointer value, they compare equal.

— If only one of two pointers to members is the null member pointer value, they compare unequal.

— If either is a pointer to a virtual member function, the result is unspecified.

— If one refers to a member of class `C1` and the other refers to a member of a different class `C2`, where neither is a base class of the other, the result is unspecified. [*Example:*

```
struct A {};
struct B : A { int x; };
struct C : A { int x; };

int A::*bx = (int(A::*))&B::x;
int A::*cx = (int(A::*))&C::x;

bool b1 = (bx == cx);    // unspecified
```

— *end example*]

— Otherwise, two pointers to members compare equal if they would refer to the same member of the same most derived object (1.319) or the same subobject if indirection with a hypothetical object of the associated class type were performed, otherwise they compare unequal. [*Example:*

```
struct B {
  int f();
};
struct L : B { };
struct R : B { };
struct D : L, R { };

int (B::*pb)() = &B::f;
int (L::*pl)() = pb;
int (R::*pr)() = pb;
int (D::*pdl)() = pl;
int (D::*pdr)() = pr;
bool x = (pdl == pdr);          // false
bool y = (pb == pl);            // true
```

— *end example*]

1 Two operands of type `std::nullptr_t` or one operand of type `std::nullptr_t` and the other a null pointer constant compare equal.

2 If two operands compare equal, the result is `true` for the `==` operator and `false` for the `!=` operator. If two operands compare unequal, the result is `false` for the `==` operator and `true` for the `!=` operator. Otherwise, the result of each of the operators is unspecified.

3

If both operands are of arithmetic or enumeration type, the usual arithmetic conversions are performed on both operands; each of the operators shall yield `true` if the specified relationship is true and `false` if it is false.

5.3274 Bitwise AND operator [expr.bit.and]

and-expression:
equality-expression
and-expression `&` *equality-expression*

1

The usual arithmetic conversions are performed; the result is the bitwise AND function of the operands. The operator applies only to integral or unscoped enumeration operands.

5.3279 Bitwise exclusive OR operator [expr.xor]

exclusive-or-expression:
and-expression
exclusive-or-expression `^` *and-expression*

1

The usual arithmetic conversions are performed; the result is the bitwise exclusive OR function of the operands. The operator applies only to integral or unscoped enumeration operands.

5.3282 Bitwise inclusive OR operator [expr.or]

inclusive-or-expression:
exclusive-or-expression
inclusive-or-expression `|` *exclusive-or-expression*

1

The usual arithmetic conversions are performed; the result is the bitwise inclusive OR function of its operands. The operator applies only to integral or unscoped enumeration operands.

5.3286 Logical AND operator [expr.log.and]

logical-and-expression:
inclusive-or-expression
logical-and-expression `&&` *inclusive-or-expression*

1

The `&&` operator groups left-to-right. The operands are both contextually converted to `bool` (Clause 4). The result is `true` if both operands are `true` and `false` otherwise. Unlike `&`, `&&` guarantees left-to-right evaluation: the second operand is not evaluated if the first operand is `false`.

1

The result is a `bool`. If the second expression is evaluated, every value computation and side effect associated with the first expression is sequenced before every value computation and side effect associated with the second expression.

5.3322 Logical OR operator [expr.log.or]

logical-or-expression:
logical-and-expression
logical-or-expression `||` *logical-and-expression*

1

The `||` operator groups left-to-right. The operands are both contextually converted to `bool` (Clause 4). It returns `true` if either of its operands is `true`, and `false` otherwise. Unlike `|`, `||` guarantees left-to-right evaluation; moreover, the second operand is not evaluated if the first operand evaluates to `true`.

1 The result is a `bool`. If the second expression is evaluated, every value computation and side effect associated with the first expression is sequenced before every value computation and side effect associated with the second expression.

5.3337 Conditional operator [expr.cond]

conditional-expression:
logical-or-expression
logical-or-expression `?` *expression* `:` *assignment-expression*

1

Conditional expressions group right-to-left. The first expression is contextually converted to `bool` (Clause 4). It is evaluated and if it is `true`, the result of the conditional expression is the value of the second expression, otherwise that of the third expression. Only one of the second and third expressions is evaluated. Every value computation and side effect associated with the first expression is sequenced before every value computation and side effect associated with the second or third expression.

1

If either the second or the third operand has type `void`, one of the following shall hold:

— The second or the third operand (but not both) is a (possibly parenthesized) *throw-expression* (15.7); the result is of the type and value category of the other.

— Both the second and the third operands have type `void`; the result is of type `void` and is a prvalue. [*Note:* This includes the case where both operands are *throw-expressions*. — *end note*]

1

Otherwise, if the second and third operand have different types and either has (possibly cv-qualified) class type, or if both are glvalues of the same value category and the same type except for cv-qualification, an attempt is made to convert each of those operands to the type of the other. The process for determining whether an operand expression `E1` of type `T1` can be converted to match an operand expression `E2` of type `T2` is defined as follows:

— If `E2` is an lvalue: `E1` can be converted to match `E2` if `E1` can be implicitly converted (Clause 4) to the type "lvalue reference to `T2`", subject to the constraint that in the conversion the reference must bind directly (8.2156.1) to an lvalue.

— If `E2` is an xvalue: `E1` can be converted to match `E2` if `E1` can be implicitly converted to the type "rvalue reference to `T2`", subject to the constraint that the reference must bind directly.

— If `E2` is a prvalue or if neither of the conversions above can be done and at least one of the operands has (possibly cv-qualified) class type:
  — if `E1` and `E2` have class type, and the underlying class types are the same or one is a base class of the other: `E1` can be converted to match `E2` if the class of `T2` is the same type as, or a base class of, the class of `T1`, and the cv-qualification of `T2` is the same cv-qualification as, or a greater cv-qualification than, the cv-qualification of `T1`. If the conversion is applied, `E1` is changed to a prvalue of type `T2` by copy-initializing a temporary of type `T2` from `E1` and using that temporary as the converted operand.
  — Otherwise (i.e., if `E1` or `E2` has a nonclass type, or if they both have class types but the underlying classes are not either the same or one a base class of the other): `E1` can be converted to match `E2` if `E1` can be implicitly converted to the type that expression `E2` would have if `E2` were converted to a prvalue (or the type it has, if `E2` is a prvalue).

Using this process, it is determined whether the second operand can be converted to match the third operand, and whether the third operand can be converted to match the second operand. If both can be converted, or one can be converted but the conversion is ambiguous, the program is ill-formed. If neither can be converted, the operands are left unchanged and further checking is performed as described below. If exactly one conversion is possible, that conversion is applied to the chosen operand and the converted operand is used in place of the original operand for the remainder of this section.

1

If the second and third operands are glvalues of the same value category and have the same type, the result is of that type and value category and it is a bit-field if the second or the third operand is a bit-field, or if both are bit-fields.

1

Otherwise, the result is a prvalue. If the second and third operands do not have the same type, and either has (possibly cv-qualified) class type, overload resolution is used to determine the conversions (if any) to be applied to the operands (13.264.0.1, 13.1258). If the overload resolution fails, the program is ill-formed. Otherwise, the conversions thus determined are applied, and the converted operands are used in place of the original operands for the remainder of this section.

1

Lvalue-to-rvalue (4.127), array-to-pointer (4.180), and function-to-pointer (4.220) standard conversions are performed on the second and third operands. After those conversions, one of the following shall hold:

— The second and third operands have the same type; the result is of that type. If the operands have class type, the result is a prvalue temporary of the result type, which is copy-initialized from either the second operand or the third operand depending on the value of the first operand.

— The second and third operands have arithmetic or enumeration type; the usual arithmetic conversions are performed to bring them to a common type, and the result is of that type.

— One or both of the second and third operands have pointer type; pointer conversions (4.579) and qualification conversions (4.246) are performed to bring them to their composite pointer type (Clause 5). The result is of the composite pointer type.

— One or both of the second and third operands have pointer to member type; pointer to member conversions (4.710) and qualification conversions (4.246) are performed to bring them to their composite pointer type (Clause 5). The result is of the composite pointer type.

— Both the second and third operands have type `std::nullptr_t` or one has that type and the other is a null pointer constant. The result is of type `std::nullptr_t`.

5.3483 Assignment and compound assignment operators [expr.ass]

1

The assignment operator (`=`) and the compound assignment operators all group right-to-left. All require a modifiable lvalue as their left operand and return an lvalue referring to the left operand. The result in all cases is a bit-field if the left operand is a bit-field. In all cases, the assignment is sequenced after the value computation of the right and left operands, and before the value computation of the assignment expression. With respect to an indeterminately-sequenced function call, the operation of a compound assignment is a single evaluation. [*Note:* Therefore, a function call shall not intervene between the lvalue-to-rvalue conversion and the side effect associated with any single compound assignment operator. — *end note*]

assignment-expression:
  conditional-expression
  logical-or-expression assignment-operator initializer-clause
  throw-expression

assignment-operator: one of
  `= *= /= %= += -= >>= <<= &= ^= |=`

1

In simple assignment (`=`), the value of the expression replaces that of the object referred to by the left operand.

1

If the left operand is not of class type, the expression is implicitly converted (Clause 4) to the cv-unqualified type of the left operand.

1

If the left operand is of class type, the class shall be complete. Assignment to objects of a class is defined by the copy/move assignment operator (12.1577, 13.1178.1).

1 [*Note:* For class objects, assignment is not in general the same as initialization (8.821, 12.30, 12.1011, 12.1577). — *end note*]

2

When the left operand of an assignment operator denotes a reference to `T`, the operation assigns to the object of type `T` denoted by the reference.

1

The behavior of an expression of the form `E1` *op* `= E2` is equivalent to `E1 = E1` *op* `E2` except that `E1` is evaluated only once. In `+=` and `-=`, `E1` shall either have arithmetic type or be a pointer to a possibly cv-qualified completely-defined object type. In all other cases, `E1` shall have arithmetic type.

1

If the value being stored in an object is accessed from another object that overlaps in any way the storage of the first object, then the overlap shall be exact and the two objects shall have the same type, otherwise the behavior is undefined. [*Note:* This restriction applies to the relationship between the left and right sides of the assignment operation; it is not a statement about how the target of the assignment may be aliased in general. See 3.3673. — *end note*]

1

A *braced-init-list* may appear on the right-hand side of

— an assignment to a scalar, in which case the initializer list shall have at most a single element. The meaning of `x={v}`, where `T` is the scalar type of the expression `x`, is that of `x=T{v}`. The meaning of `x={}` is `x=T{}`.

— an assignment to an object of class type, in which case the initializer list is passed as the argument to the assignment operator function selected by overload resolution (13.1178.1, 13.88).

[*Example:*

```
complex<double> z;
z = { 1,2 };              // meaning z.operator=({1,2})
z += { 1, 2 };            // meaning z.operator+=({1,2})
int a, b;
a = b = { 1 };            // meaning a=b=1;
a = { 1 } = b;            // syntax error
```

— *end example*]

5.3602 Comma operator [expr.comma]

1

The comma operator groups left-to-right.

expression:
  assignment-expression
  expression **,** *assignment-expression*

A pair of expressions separated by a comma is evaluated left-to-right; the left expression is a discarded-value expression (Clause 5).[86] Every value computation and side effect associated with the left expression is sequenced before every value computation and side effect associated with the right expression. The type and value of the result are the type and value of the right operand; the result is of the same value category as its right operand, and is a bit-field if its right operand is a glvalue and a bit-field. If the value of the right operand is a temporary (12.306), the result is that temporary.

1

In contexts where comma is given a special meaning, [*Example:* in lists of arguments to functions (5.955.1) and lists of initializers (8.821) — *end example*] the comma operator as described in Clause 5 can appear only in parentheses. [*Example:*

```
f(a, (t=3, t+2), c);
```

has three arguments, the second of which has the value `5`. — *end example*]

5.3643 Constant expressions [expr.const]

1

Certain contexts require expressions that satisfy additional requirements as detailed in this sub-clause; other contexts have different semantics depending on whether or not an expression satisfies these requirements. Expressions that satisfy these requirements are called *constant expressions*. [*Note:* Constant expressions can be evaluated during translation. — *end note*]

constant-expression:
    conditional-expression

1

A *conditional-expression* `e` is a *core constant expression* unless the evaluation of `e`, following the rules of the abstract machine (1.430), would evaluate one of the following expressions:

— `this` (5.398.1), except in a `constexpr` function or a `constexpr` constructor that is being evaluated as part of `e`;

— an invocation of a function other than a `constexpr` constructor for a literal class, a `constexpr` function, or an implicit invocation of a trivial destructor (12.807) [*Note:* Overload resolution (13.88) is applied as usual — *end note*];

— an invocation of an undefined `constexpr` function or an undefined `constexpr` constructor;

— an expression that would exceed the implementation-defined limits (see Annex B);

— an operation that would have undefined behavior [*Note:* including, for example, signed integer overflow (Clause 5), certain pointer arithmetic (5.2882), division by zero (5.2851), or certain shift operations (5.3085) — *end note*];

— a *lambda-expression* (5.509.1);

— an lvalue-to-rvalue conversion (4.127) unless it is applied to

  — a non-volatile glvalue of integral or enumeration type that refers to a non-volatile const object with a preceding initialization, initialized with a constant expression [*Note:* a string literal (2.453.1) corresponds to an array of such objects. — *end note*], or

86) However, an invocation of an overloaded comma operator is an ordinary function call; hence, the evaluations of its argument expressions are unsequenced relative to one another (see 1.430).

  - a non-volatile glvalue that refers to a non-volatile object defined with `constexpr`, or that refers to a non-mutable sub-object of such an object, or
  - a non-volatile glvalue of literal type that refers to a non-volatile object whose lifetime began within the evaluation of `e`;

- an lvalue-to-rvalue conversion (4.127) or modification (5.3483, 5.1264.1, 5.2054.1) that is applied to a glvalue that refers to a non-active member of a union or a subobject thereof;
- an *id-expression* that refers to a variable or data member of reference type unless the reference has a preceding initialization and either
  - it is initialized with a constant expression or
  - it is a non-static data member of an object whose lifetime began within the evaluation of `e`;
- in a *lambda-expression*, a reference to `this` or to a variable with automatic storage duration defined outside that *lambda-expression*, where the reference would be an odr-use (3.91, 5.509.1);
- a conversion from type *cv* `void *` to a pointer-to-object type;
- a dynamic cast (5.1295.1);
- a `reinterpret_cast` (5.1644.1);
- a pseudo-destructor call (5.1181.1);
- modification of an object (5.3483, 5.1264.1, 5.2054.1) unless it is applied to a non-volatile lvalue of literal type that refers to a non-volatile object whose lifetime began within the evaluation of `e`;
- a typeid expression (5.1393.1) whose operand is a glvalue of a polymorphic class type;
- a *new-expression* (5.2221.1);
- a *delete-expression* (5.2455.1);
- a relational (5.3203) or equality (5.3247) operator where the result is unspecified; or
- a *throw-expression* (15.7).

[*Example:*

```
int x;                              // not constant
struct A {
  constexpr A(bool b) : m(b?42:x) { }
  int m;
};
constexpr int v = A(true).m;        // OK: constructor call initializes
                                    // m with the value 42
constexpr int w = A(false).m;       // error: initializer for m is
                                    // x, which is non-constant

constexpr int f1(int k) {
  constexpr int x = k;              // error: x is not initialized by a
                                    // constant expression because lifetime of k
                                    // began outside the initializer of x
  return x;
}
constexpr int f2(int k) {
  int x = k;                        // OK: not required to be a constant expression
```

```
                                        // because x is not constexpr
  return x;
}

constexpr int incr(int &n) {
  return ++n;
}
constexpr int g(int k) {
  constexpr int x = incr(k);            // error: incr(k) is not a core constant
                                        // expression because lifetime of k
                                        // began outside the expression incr(k)
  return x;
}
constexpr int h(int k) {
  int x = incr(k);                      // OK: incr(k) is not required to be a core
                                        // constant expression
  return x;
}
constexpr int y = h(1);                 // OK: initializes y with the value 2
                                        // h(1) is a core constant expression because
                                        // the lifetime of k begins inside h(1)
```

— *end example*]

1

An *integral constant expression* is an expression of integral or unscoped enumeration type, implicitly converted to a prvalue, where the converted expression is a core constant expression. [*Note:* Such expressions may be used as array bounds (8.251.1, 5.2221.1), as bit-field lengths (9.921), as enumerator initializers if the underlying type is not fixed (7.1001), and as alignments (7.1556.1). — *end note*] A *converted constant expression* of type `T` is an expression, implicitly converted to a prvalue of type `T`, where the converted expression is a core constant expression and the implicit conversion sequence contains only user-defined conversions, lvalue-to-rvalue conversions (4.127), integral promotions (4.262), and integral conversions (4.397) other than narrowing conversions (8.2435.1). [*Note:* such expressions may be used in `new` expressions (5.2221.1), as case expressions (6.72.1), as enumerator initializers if the underlying type is fixed (7.1001), as array bounds (8.251.1), and as integral or enumeration non-type template arguments (14.326). — *end note*]

1

A *constant expression* is either a glvalue core constant expression whose value refers to an object with static storage duration or to a function, or a prvalue core constant expression whose value is an object where, for that object and its subobjects:

— each non-static data member of reference type refers to an object with static storage duration or to a function, and

— if the object or subobject is of pointer type, it contains the address of an object with static storage duration, the address past the end of such an object (5.2882), the address of a function, or a null pointer value.

1

[*Note:* Since this International Standard imposes no restrictions on the accuracy of floating-point operations, it is unspecified whether the evaluation of a floating-point expression during translation yields the same result as the evaluation of the same expression (or the same operations on the same values) during program

execution.[87] [*Example:*

```
bool f() {
    char array[1 + int(1 + 0.2 - 0.1 - 0.1)];  // Must be evaluated during translation
    int size = 1 + int(1 + 0.2 - 0.1 - 0.1);   // May be evaluated at runtime
    return sizeof(array) == size;
}
```

It is unspecified whether the value of `f()` will be `true` or `false`. — *end example*] — *end note*]

1

If an expression of literal class type is used in a context where an integral constant expression is required, then that expression is contextually implicitly converted (Clause 4) to an integral or unscoped enumeration type and the selected conversion function shall be `constexpr`. [*Example:*

```
struct A {
  constexpr A(int i) : val(i) { }
  constexpr operator int() const { return val; }
  constexpr operator long() const { return 43; }
private:
  int val;
};
template<int> struct X { };
constexpr A a = 42;
X<a> x;             // OK: unique conversion to int
int ary[a];         // error: ambiguous conversion
```

— *end example*]

87) Nonetheless, implementations are encouraged to provide consistent results, irrespective of whether the evaluation was performed during translation and/or during program execution.

6 Statements [stmt.stmt]

1

Except as indicated, statements are executed in sequence.

statement:
- *labeled-statement*
- *attribute-specifier-seq*$_{opt}$ *expression-statement*
- *attribute-specifier-seq*$_{opt}$ *compound-statement*
- *attribute-specifier-seq*$_{opt}$ *selection-statement*
- *attribute-specifier-seq*$_{opt}$ *iteration-statement*
- *attribute-specifier-seq*$_{opt}$ *jump-statement*
- *declaration-statement*
- *attribute-specifier-seq*$_{opt}$ *try-block*

The optional *attribute-specifier-seq* appertains to the respective statement.

6.3 Labeled statement [stmt.label]

1

A statement can be labeled.

labeled-statement:
- *attribute-specifier-seq*$_{opt}$ *identifier* `:` *statement*
- *attribute-specifier-seq*$_{opt}$`case` *constant-expression* `:` *statement*
- *attribute-specifier-seq*$_{opt}$`default :` *statement*

The optional *attribute-specifier-seq* appertains to the label. An identifier label declares the identifier. The only use of an identifier label is as the target of a `goto`. The scope of a label is the function in which it appears. Labels shall not be redeclared within a function. A label can be used in a `goto` statement before its definition. Labels have their own name space and do not interfere with other identifiers.

1

Case labels and default labels shall occur only in switch statements.

6.9 Expression statement [stmt.expr]

1

Expression statements have the form

expression-statement:
- *expression*$_{opt}$ `;`

The expression is a discarded-value expression (Clause 5). All side effects from an expression statement are completed before the next statement is executed. An expression statement with the expression missing is called a null statement. [*Note:* Most statements are expression statements — usually assignments or function calls. A null statement is useful to carry a label just before the `}` of a compound statement and to supply a null body to an iteration statement such as a `while` statement (6.144.1). — *end note*]

6.15 Compound statement or block [stmt.block]

1

So that several statements can be used where one is expected, the compound statement (also, and equivalently, called "block") is provided.

compound-statement:
  { *statement-seq*$_{opt}$}
statement-seq:
  statement
  statement-seq statement

A compound statement defines a block scope (3.464). [*Note:* A declaration is a statement (6.419). — *end note*]

6.23 Selection statements [stmt.select]

1

Selection statements choose one of several flows of control.

selection-statement:
  `if` (*condition*) *statement*
  `if` (*condition*) *statement* `else` *statement*
  `switch` (*condition*) *statement*
condition:
  expression
  attribute-specifier-seq$_{opt}$ *decl-specifier-seq declarator* `=` *initializer-clause*
  attribute-specifier-seq$_{opt}$ *decl-specifier-seq declarator braced-init-list*

See 8.56 for the optional *attribute-specifier-seq* in a condition. In Clause 6, the term *substatement* refers to the contained statement or statements that appear in the syntax notation. The substatement in a *selection-statement* (each substatement, in the `else` form of the `if` statement) implicitly defines a block scope (3.464). If the substatement in a selection-statement is a single statement and not a *compound-statement,* it is as if it was rewritten to be a compound-statement containing the original substatement. [*Example:*

```
if (x)
  int i;
```

can be equivalently rewritten as

```
if (x) {
  int i;
}
```

Thus after the `if` statement, `i` is no longer in scope. — *end example*]

1

The rules for conditions apply both to *selection-statement*s and to the `for` and `while` statements (6.137). The declarator shall not specify a function or an array. The *decl-specifier-seq* shall not define a class or enumeration. If the `auto` *type-specifier* appears in the *decl-specifier-seq*, the type of the identifier being declared is deduced from the initializer as described in 7.832.0.1.

1

A name introduced by a declaration in a condition (either introduced by the *decl-specifier-seq* or the declarator of the condition) is in scope from its point of declaration until the end of the substatements controlled by the condition. If the name is re-declared in the outermost block of a substatement controlled by the condition, the declaration that re-declares the name is ill-formed. [*Example:*

```
if (int x = f()) {
  int x;            // ill-formed, redeclaration of x
}
else {
  int x;            // ill-formed, redeclaration of x
}
```

— *end example*]

1

The value of a condition that is an initialized declaration in a statement other than a `switch` statement is the value of the declared variable contextually converted to `bool` (Clause 4). If that conversion is ill-formed, the program is ill-formed. The value of a condition that is an initialized declaration in a `switch` statement is the value of the declared variable if it has integral or enumeration type, or of that variable implicitly converted to integral or enumeration type otherwise. The value of a condition that is an expression is the value of the expression, contextually converted to `bool` for statements other than `switch`; if that conversion is ill-formed, the program is ill-formed. The value of the condition will be referred to as simply "the condition" where the usage is unambiguous.

1 If a condition can be syntactically resolved as either an expression or the declaration of a block-scope name, it is interpreted as a declaration.

2

In the *decl-specifier-seq* of a *condition*, each *decl-specifier* shall be either a *type-specifier* or `constexpr`.

6.66.1 The `if` statement [stmt.if]

1

If the condition (6.23) yields `true` the first substatement is executed. If the `else` part of the selection statement is present and the condition yields `false`, the second substatement is executed. If the first substatement is reached via a label, the condition is not evaluated and the second substatement is not executed. In the second form of `if` statement (the one including `else`), if the first substatement is also an `if` statement then that inner `if` statement shall contain an `else` part.[88]

6.72.1 The `switch` statement [stmt.switch]

1

The `switch` statement causes control to be transferred to one of several statements depending on the value of a condition.

1

The condition shall be of integral type, enumeration type, or class type. If of class type, the condition is contextually implicitly converted (Clause 4) to an integral or enumeration type. If the (possibly converted) type is subject to integral promotions (4.262), the condition is converted to the promoted type. Any statement within the `switch` statement can be labeled with one or more case labels as follows:

`case` *constant-expression* `:`

where the *constant-expression* shall be a converted constant expression (5.3643) of the adjusted type of the switch condition. No two of the case constants in the same switch shall have the same value after conversion.

1 There shall be at most one label of the form

```
default :
```

within a `switch` statement.

2

Switch statements can be nested; a `case` or `default` label is associated with the smallest switch enclosing it.

1

88) In other words, the `else` is associated with the nearest un-elsed `if`.

When the `switch` statement is executed, its condition is evaluated and compared with each case constant. If one of the case constants is equal to the value of the condition, control is passed to the statement following the matched case label. If no case constant matches the condition, and if there is a `default` label, control passes to the statement labeled by the default label. If no case matches and if there is no `default` then none of the statements in the switch is executed.

1

`case` and `default` labels in themselves do not alter the flow of control, which continues unimpeded across such labels. To exit from a switch, see `break`, 6.263.1. [*Note:* Usually, the substatement that is the subject of a switch is compound and `case` and `default` labels appear on the top-level statements contained within the (compound) substatement, but this is not required. Declarations can appear in the substatement of a *switch-statement*. — *end note*]

6.137 Iteration statements [stmt.iter]

1

Iteration statements specify looping.

iteration-statement:
- `while` `(` *condition* `)` *statement*
- `do` *statement* `while` `(` *expression* `)` `;`
- `for` `(` *for-init-statement condition*$_{opt}$`;` *expression*$_{opt}$`)` *statement*
- `for` `(` *for-range-declaration* `:` *for-range-initializer* `)` *statement*

for-init-statement:
- *expression-statement*
- *simple-declaration*

for-range-declaration:
- *attribute-specifier-seq*$_{opt}$ *decl-specifier-seq declarator*

for-range-initializer:
- *expression*
- *braced-init-list*

See 8.56 for the optional *attribute-specifier-seq* in a *for-range-declaration*. [*Note:* A *for-init-statement* ends with a semicolon. — *end note*]

1

The substatement in an *iteration-statement* implicitly defines a block scope (3.464) which is entered and exited each time through the loop.

If the substatement in an iteration-statement is a single statement and not a *compound-statement,* it is as if it was rewritten to be a compound-statement containing the original statement. [*Example:*

```
while (--x >= 0)
  int i;
```

can be equivalently rewritten as

```
while (--x >= 0) {
  int i;
}
```

1 Thus after the `while` statement, `i` is no longer in scope. — *end example*]

2 [*Note:* The requirements on conditions in iteration statements are described in 6.23. — *end note*]

6.144.1 The `while` statement [stmt.while]

1

In the `while` statement the substatement is executed repeatedly until the value of the condition (6.23) becomes `false`. The test takes place before each execution of the substatement.

1

When the condition of a `while` statement is a declaration, the scope of the variable that is declared extends from its point of declaration (3.536.1) to the end of the `while` statement. A `while` statement of the form

```
while (T t = x) statement
```

is equivalent to

```
label:
{                       // start of condition scope
  T t = x;
  if (t) {
    statement
    goto label;
  }
}                       // end of condition scope
```

The variable created in a condition is destroyed and created with each iteration of the loop. [*Example:*

```
struct A {
  int val;
  A(int i) : val(i) { }
  ~A() { }
  operator bool() { return val != 0; }
};
int i = 1;
while (A a = i) {
  // ...
  i = 0;
}
```

In the while-loop, the constructor and destructor are each called twice, once for the condition that succeeds and once for the condition that fails. — *end example*]

6.149.1 The `do` statement [stmt.do]

1 The expression is contextually converted to `bool` (Clause 4); if that conversion is ill-formed, the program is ill-formed.

2

In the `do` statement the substatement is executed repeatedly until the value of the expression becomes `false`. The test takes place after each execution of the statement.

6.150.1 The `for` statement [stmt.for]

1

The `for` statement

`for (` *for-init-statement condition*$_{opt}$ `;` *expression*$_{opt}$ `)` *statement*

is equivalent to

```
{
    for-init-statement
    while ( condition ) {
        statement
        expression ;
    }
}
```

except that names declared in the *for-init-statement* are in the same declarative region as those declared in the condition, and except that a `continue` in statement (not enclosed in another iteration statement) will execute expression before re-evaluating condition. [*Note:* Thus the first statement specifies initialization for the loop; the condition (6.23) specifies a test, made before each iteration, such that the loop is exited when the condition becomes `false`; the expression often specifies incrementing that is done after each iteration. — *end note*]

1

Either or both of the condition and the expression can be omitted. A missing condition makes the implied `while` clause equivalent to `while(true)`.

1

If the *for-init-statement* is a declaration, the scope of the name(s) declared extends to the end of the `for` statement. [*Example:*

```
int i = 42;
int a[10];

for (int i = 0; i < 10; i++)
  a[i] = i;

int j = i;          // j = 42
```

— *end example*]

6.165.1 The range-based `for` statement [stmt.ranged]

1

For a range-based `for` statement of the form

`for` (*for-range-declaration : expression*) *statement*

let *range-init* be equivalent to the *expression* surrounded by parentheses[89]

(*expression*)

and for a range-based `for` statement of the form

`for` (*for-range-declaration* : *braced-init-list*) *statement*

let *range-init* be equivalent to the *braced-init-list*. In each case, a range-based `for` statement is equivalent to

```
{
  auto && __range = range-init;
  for ( auto __begin = begin-expr,
            __end = end-expr;
        __begin != __end;
        ++__begin ) {
```

89) this ensures that a top-level comma operator cannot be reinterpreted as a delimiter between *init-declarator*s in the declaration of `__range`.

```
    for-range-declaration = *__begin;
    statement
  }
}
```

where `__range`, `__begin`, and `__end` are variables defined for exposition only, and `_RangeT` is the type of the expression, and *begin-expr* and *end-expr* are determined as follows:

— if `_RangeT` is an array type, *begin-expr* and *end-expr* are `__range` and `__range + __bound`, respectively, where `__bound` is the array bound. If `_RangeT` is an array of unknown size or an array of incomplete type, the program is ill-formed;

— if `_RangeT` is a class type, the *unqualified-id*s `begin` and `end` are looked up in the scope of class `_RangeT` as if by class member access lookup (3.1423.1), and if either (or both) finds at least one declaration, *begin-expr* and *end-expr* are `__range.begin()` and `__range.end()`, respectively;

— otherwise, *begin-expr* and *end-expr* are `begin(__range)` and `end(__range)`, respectively, where `begin` and `end` are looked up in the associated namespaces (3.1092.1). [*Note:* Ordinary unqualified lookup (3.952.1) is not performed. — *end note*]

[*Example:*

```
int array[5] = { 1, 2, 3, 4, 5 };
for (int& x : array)
  x *= 2;
```

— *end example*]

1

In the *decl-specifier-seq* of a *for-range-declaration*, each *decl-specifier* shall be either a *type-specifier* or `constexpr`. The *decl-specifier-seq* shall not define a class or enumeration.

6.236 Jump statements [stmt.jump]

1 Jump statements unconditionally transfer control.

jump-statement:
- `break ;`
- `continue ;`
- `return` *expression*$_{opt}$ `;`
- `return` *braced-init-list* `;`
- `goto` *identifier* `;`

2

On exit from a scope (however accomplished), objects with automatic storage duration (3.2301.1) that have been constructed in that scope are destroyed in the reverse order of their construction. [*Note:* For temporaries, see 12.306. — *end note*] Transfer out of a loop, out of a block, or back past an initialized variable with automatic storage duration involves the destruction of objects with automatic storage duration that are in scope at the point transferred from but not at the point transferred to. (See 6.419 for transfers into blocks). [*Note:* However, the program can be terminated (by calling `std::exit()` or `std::abort()` (18.127), for example) without destroying class objects with automatic storage duration. — *end note*]

6.263.1 The `break` statement [stmt.break]

1

The `break` statement shall occur only in an *iteration-statement* or a `switch` statement and causes termination of the smallest enclosing *iteration-statement* or `switch` statement; control passes to the statement following the terminated statement, if any.

6.270.1 The `continue` statement [stmt.cont]

1 The `continue` statement shall occur only in an *iteration-statement* and causes control to pass to the loop-continuation portion of the smallest enclosing *iteration-statement*, that is, to the end of the loop. More precisely, in each of the statements

```
while (foo) {
  {
    // ...
  }
contin: ;
}
```

```
do {
  {
    // ...
  }
contin: ;
} while (foo);
```

```
for (;;) {
  {
    // ...
  }
contin: ;
}
```

a `continue` not contained in an enclosed iteration statement is equivalent to `goto contin`.

6.270.2 The `return` statement [stmt.return]

1

A function returns to its caller by the `return` statement.

1

A return statement with neither an *expression* nor a *braced-init-list* can be used only in functions that do not return a value, that is, a function with the return type *cv* `void`, a constructor (12.30), or a destructor (12.807). A return statement with an expression of non-void type can be used only in functions returning a value; the value of the expression is returned to the caller of the function. The value of the expression is implicitly converted to the return type of the function in which it appears. A return statement can involve the construction and copy or move of a temporary object (12.306). [*Note:* A copy or move operation associated with a return statement may be elided or considered as an rvalue for the purpose of overload resolution in selecting a constructor (12.1577). — *end note*] A return statement with a *braced-init-list* initializes the object or reference to be returned from the function by copy-list-initialization (8.2435.1) from the specified initializer list. [*Example:*

```
std::pair<std::string,int> f(const char* p, int x) {
  return {p,x};
}
```

— *end example*]

Flowing off the end of a function is equivalent to a `return` with no value; this results in undefined behavior in a value-returning function.

1

A return statement with an expression of type `void` can be used only in functions with a return type of *cv* `void`; the expression is evaluated just before the function returns to its caller.

6.413.1 The `goto` statement [stmt.goto]

1

The `goto` statement unconditionally transfers control to the statement labeled by the identifier. The identifier shall be a label (6.3) located in the current function.

6.419 Declaration statement [stmt.dcl]

1

A declaration statement introduces one or more new identifiers into a block; it has the form

declaration-statement:
block-declaration

If an identifier introduced by a declaration was previously declared in an outer block, the outer declaration is hidden for the remainder of the block, after which it resumes its force.

1

Variables with automatic storage duration (3.2301.1) are initialized each time their *declaration-statement* is executed. Variables with automatic storage duration declared in the block are destroyed on exit from the block (6.236).

1

It is possible to transfer into a block, but not in a way that bypasses declarations with initialization. A program that jumps[90] from a point where a variable with automatic storage duration is not in scope to a point where it is in scope is ill-formed unless the variable has scalar type, class type with a trivial default constructor and a trivial destructor, a cv-qualified version of one of these types, or an array of one of the preceding types and is declared without an initializer (8.821). [*Example:*

```
void f() {
  // ...
  goto lx;          // ill-formed: jump into scope of a
  // ...
ly:
  X a = 1;
  // ...
lx:
  goto ly;          // OK, jump implies destructor
                    // call for a followed by construction
                    // again immediately following label ly
}
```

— *end example*]

1

The zero-initialization (8.821) of all block-scope variables with static storage duration (3.2242.1) or thread storage duration (3.2291.1) is performed before any other initialization takes place. Constant initialization (3.1937.1) of a block-scope entity with static storage duration, if applicable, is performed before its block is first entered. An implementation is permitted to perform early initialization of other block-scope variables with static or thread storage duration under the same conditions that an implementation is permitted to statically initialize a variable with static or thread storage duration in namespace scope (3.1937.1). Otherwise such a variable is initialized the first time control passes through its declaration; such a variable is considered initialized upon the completion of its initialization. If the initialization exits by throwing an exception, the initialization is not complete, so it will be tried again the next time control enters the declaration. If control enters the declaration concurrently while the variable is being initialized, the concurrent execution shall wait for completion of the initialization.[91] If control re-enters the declaration recursively while the variable is being initialized, the behavior is undefined. [*Example:*

```
int foo(int i) {
  static int s = foo(2*i);      // recursive call - undefined
  return i+1;
}
```

— *end example*]

1

90) The transfer from the condition of a `switch` statement to a `case` label is considered a jump in this respect.
91) The implementation must not introduce any deadlock around execution of the initializer.

The destructor for a block-scope object with static or thread storage duration will be executed if and only if it was constructed. [*Note:* 3.2200.1 describes the order in which block-scope objects with static and thread storage duration are destroyed. — *end note*]

6.556 Ambiguity resolution [stmt.ambig]

1

There is an ambiguity in the grammar involving *expression-statement*s and *declaration*s: An *expression-statement* with a function-style explicit type conversion (5.1108.1) as its leftmost subexpression can be indistinguishable from a *declaration* where the first *declarator* starts with a `(`. In those cases the *statement* is a *declaration*.

1

[*Note:* If the *statement* cannot syntactically be a *declaration*, there is no ambiguity, so this rule does not apply. The whole *statement* might need to be examined to determine whether this is the case. This resolves the meaning of many examples. [*Example:* Assuming `T` is a *simple-type-specifier* (7.519.1),

```
T(a)->m = 7;          // expression-statement
T(a)++;               // expression-statement
T(a,5)<<c;            // expression-statement

T(*d)(int);           // declaration
T(e)[5];              // declaration
T(f) = { 1, 2 };      // declaration
T(*g)(double(3));     // declaration
```

In the last example above, `g`, which is a pointer to `T`, is initialized to `double(3)`. This is of course ill-formed for semantic reasons, but that does not affect the syntactic analysis. — *end example*]

The remaining cases are *declaration*s. [*Example:*

```
class T {
  // ...
public:
  T();
  T(int);
  T(int, int);
};
T(a);                 // declaration
T(*b)();              // declaration
T(c)=7;               // declaration
T(d),e,f=3;           // declaration
extern int h;
T(g)(h,2);            // declaration
```

— *end example*] — *end note*]

1

The disambiguation is purely syntactic; that is, the meaning of the names occurring in such a statement, beyond whether they are *type-name*s or not, is not generally used in or changed by the disambiguation. Class templates are instantiated as necessary to determine if a qualified name is a *type-name*. Disambiguation precedes parsing, and a statement disambiguated as a declaration may be an ill-formed declaration. If, during parsing, a name in a template parameter is bound differently than it would be bound during a trial parse, the program is ill-formed. No diagnostic is required. [*Note:* This can occur only when the name is declared earlier in the declaration. — *end note*] [*Example:*

```
struct T1 {
  T1 operator()(int x) { return T1(x); }
  int operator=(int x) { return x; }
  T1(int) { }
};
struct T2 { T2(int){ } };
int a, (*(*b)(T2))(int), c, d;

void f() {
  // disambiguation requires this to be parsed as a declaration:
  T1(a) = 3,
  T2(4),                        // T2 will be declared as
  (*(*b)(T2(c)))(int(d));       // a variable of type T1
                                // but this will not allow
                                // the last part of the
                                // declaration to parse
                                // properly since it depends
                                // on T2 being a type-name
}
```

— *end example*]

7 Declarations [dcl.dcl]

1 Declarations generally specify how names are to be interpreted. Declarations have the form

declaration-seq:
  declaration
  declaration-seq declaration

declaration:
  block-declaration
  function-definition
  template-declaration
  explicit-instantiation
  explicit-specialization
  linkage-specification
  namespace-definition
  empty-declaration
  attribute-declaration

block-declaration:
  simple-declaration
  asm-definition
  namespace-alias-definition
  using-declaration
  using-directive
  static_assert-declaration
  alias-declaration
  opaque-enum-declaration

alias-declaration:
  `using` *identifier attribute-specifier-seq*$_{opt}$ `=` *type-id* `;`

simple-declaration:
  decl-specifier-seq$_{opt}$ *init-declarator-list*$_{opt}$ `;`
  attribute-specifier-seq decl-specifier-seq$_{opt}$ *init-declarator-list* `;`

static_assert-declaration:
  `static_assert (` *constant-expression* `,` *string-literal* `) ;`

empty-declaration:
  `;`

attribute-declaration:
  attribute-specifier-seq `;`

[*Note: asm-definition*s are described in 7.1473, and *linkage-specification*s are described in 7.1477. *Function-definition*s are described in 8.706 and *template-declaration*s are described in Clause 14. *Namespace-definition*s are described in 7.1149.1, *using-declaration*s are described in 7.1311.1 and *using-directive*s are described in 7.1416.1. — *end note*]

2 The *simple-declaration*

attribute-specifier-seq$_{opt}$ *decl-specifier-seq*$_{opt}$ *init-declarator-list*$_{opt}$ `;`

is divided into three parts. Attributes are described in 7.1546. *decl-specifier*s, the principal components of a *decl-specifier-seq*, are described in 7.40. *declarator*s, the components of an *init-declarator-list*, are described in Clause 8. The *attribute-specifier-seq* in a *simple-declaration* appertains to each of the entities declared by the *declarator*s of the *init-declarator-list*. [*Note:* In the declaration for an entity, attributes appertaining to that entity may appear at the start of the declaration and after the *declarator-id* for that declaration.

— *end note*] [*Example:*

```
[[noreturn]] void f [[noreturn]] (); // OK
```

— *end example*]

3 Except where otherwise specified, the meaning of an *attribute-declaration* is implementation-defined.

4 A declaration occurs in a scope (3.464); the scope rules are summarized in 3.923. A declaration that declares a function or defines a class, namespace, template, or function also has one or more scopes nested within it. These nested scopes, in turn, can have declarations nested within them. Unless otherwise stated, utterances in Clause 7 about components in, of, or contained by a declaration or subcomponent thereof refer only to those components of the declaration that are *not* nested within scopes nested within the declaration.

5

In a *simple-declaration*, the optional *init-declarator-list* can be omitted only when declaring a class (Clause 9) or enumeration (7.1001), that is, when the *decl-specifier-seq* contains either a *class-specifier*, an *elaborated-type-specifier* with a *class-key* (9.172), or an *enum-specifier*. In these cases and whenever a *class-specifier* or *enum-specifier* is present in the *decl-specifier-seq*, the identifiers in these specifiers are among the names being declared by the declaration (as *class-name*s, *enum-name*s, or *enumerator*s, depending on the syntax). In such cases, the *decl-specifier-seq* shall introduce one or more names into the program, or shall redeclare a name introduced by a previous declaration. [*Example:*

```
enum { };           // ill-formed
typedef class { };  // ill-formed
```

— *end example*]

1

In a *static_assert-declaration* the *constant-expression* shall be a constant expression (5.3643) that can be contextually converted to `bool` (Clause 4). If the value of the expression when so converted is `true`, the declaration has no effect. Otherwise, the program is ill-formed, and the resulting diagnostic message (1.196) shall include the text of the *string-literal*, except that characters not in the basic source character set (2.154) are not required to appear in the diagnostic message. [*Example:*

```
static_assert(sizeof(long) >= 8, "64-bit code generation required for this library.");
```

— *end example*]

1 An *empty-declaration* has no effect.

2

Each *init-declarator* in the *init-declarator-list* contains exactly one *declarator-id*, which is the name declared by that *init-declarator* and hence one of the names declared by the declaration. The *type-specifiers* (7.519.1) in the *decl-specifier-seq* and the recursive *declarator* structure of the *init-declarator* describe a type (8.56), which is then associated with the name being declared by the *init-declarator*.

1

If the *decl-specifier-seq* contains the `typedef` specifier, the declaration is called a *typedef declaration* and the name of each *init-declarator* is declared to be a *typedef-name*, synonymous with its associated type (7.235.1). If the *decl-specifier-seq* contains no `typedef` specifier, the declaration is called a *function declaration* if the type associated with the name is a function type (8.360.1) and an *object declaration* otherwise.

1

Syntactic components beyond those found in the general form of declaration are added to a function declaration to make a *function-definition*. An object declaration, however, is also a definition unless it contains

the `extern` specifier and has no initializer (3.43). A definition causes the appropriate amount of storage to be reserved and any appropriate initialization (8.821) to be done.

1

Only in function declarations for constructors, destructors, and type conversions can the *decl-specifier-seq* be omitted.[92]

7.40 Specifiers [dcl.spec]

1

The specifiers that can be used in a declaration are

decl-specifier:
- *storage-class-specifier*
- *type-specifier*
- *function-specifier*
- `friend`
- `typedef`
- `constexpr`

decl-specifier-seq:
- *decl-specifier attribute-specifier-seq*$_{opt}$
- *decl-specifier decl-specifier-seq*

The optional *attribute-specifier-seq* in a *decl-specifier-seq* appertains to the type determined by the preceding *decl-specifier*s (8.56). The *attribute-specifier-seq* affects the type only for the declaration it appears in, not other declarations involving the same type.

1

If a *type-name* is encountered while parsing a *decl-specifier-seq*, it is interpreted as part of the *decl-specifier-seq* if and only if there is no previous *type-specifier* other than a *cv-qualifier* in the *decl-specifier-seq*. The sequence shall be self-consistent as described below. [*Example:*

```
typedef char* Pc;
static Pc;                      // error: name missing
```

Here, the declaration `static Pc` is ill-formed because no name was specified for the static variable of type `Pc`. To get a variable called `Pc`, a *type-specifier* (other than `const` or `volatile`) has to be present to indicate that the *typedef-name* `Pc` is the name being (re)declared, rather than being part of the *decl-specifier* sequence. For another example,

```
void f(const Pc);               // void f(char* const) (not const char*)
void g(const int Pc);           // void g(const int)
```

— *end example*]

1

[*Note:* Since `signed`, `unsigned`, `long`, and `short` by default imply `int`, a *type-name* appearing after one of those specifiers is treated as the name being (re)declared. [*Example:*

```
void h(unsigned Pc);            // void h(unsigned int)
void k(unsigned int Pc);        // void k(unsigned int)
```

— *end example*] — *end note*]

92) The "implicit int" rule of C is no longer supported.

7.49.1 Storage class specifiers [dcl.stc]

1

The storage class specifiers are

storage-class-specifier:
`register`
`static`
`thread_local`
`extern`
`mutable`

At most one *storage-class-specifier* shall appear in a given *decl-specifier-seq*, except that `thread_local` may appear with `static` or `extern`. If `thread_local` appears in any declaration of a variable it shall be present in all declarations of that entity. If a *storage-class-specifier* appears in a *decl-specifier-seq*, there can be no `typedef` specifier in the same *decl-specifier-seq* and the *init-declarator-list* of the declaration shall not be empty (except for an anonymous union declared in a named namespace or in the global namespace, which shall be declared `static` (9.815)). The *storage-class-specifier* applies to the name declared by each *init-declarator* in the list and not to any names declared by other specifiers. A *storage-class-specifier* shall not be specified in an explicit specialization (14.1921.1) or an explicit instantiation (14.1841.1) directive.

1

The `register` specifier shall be applied only to names of variables declared in a block (6.15) or to function parameters (8.706). It specifies that the named variable has automatic storage duration (3.2301.1). A variable declared without a *storage-class-specifier* at block scope or declared as a function parameter has automatic storage duration by default.

1

A `register` specifier is a hint to the implementation that the variable so declared will be heavily used. [*Note:* The hint can be ignored and in most implementations it will be ignored if the address of the variable is taken. This use is deprecated (see D.4). — *end note*]

1

The `thread_local` specifier indicates that the named entity has thread storage duration (3.2291.1). It shall be applied only to the names of variables of namespace or block scope and to the names of static data members. When `thread_local` is applied to a variable of block scope the *storage-class-specifier* `static` is implied if no other *storage-class-specifier* appears in the *decl-specifier-seq*.

1

The `static` specifier can be applied only to names of variables and functions and to anonymous unions (9.815). There can be no `static` function declarations within a block, nor any `static` function parameters. A `static` specifier used in the declaration of a variable declares the variable to have static storage duration (3.2242.1), unless accompanied by the `thread_local` specifier, which declares the variable to have thread storage duration (3.2291.1). A `static` specifier can be used in declarations of class members; 9.559 describes its effect. For the linkage of a name declared with a `static` specifier, see 3.1459.

1

The `extern` specifier can be applied only to the names of variables and functions. The `extern` specifier cannot be used in the declaration of class members or function parameters. For the linkage of a name declared with an `extern` specifier, see 3.1459. [*Note:* The `extern` keyword can also be used in *explicit-instantiation*s and *linkage-specification*s, but it is not a *storage-class-specifier* in such contexts. — *end note*]

1

The linkages implied by successive declarations for a given entity shall agree. That is, within a given scope, each declaration declaring the same variable name or the same overloading of a function name shall imply

the same linkage. Each function in a given set of overloaded functions can have a different linkage, however. [*Example:*

```
static char* f();                // f() has internal linkage
char* f()                        // f() still has internal linkage
  { /* ... */ }

char* g();                       // g() has external linkage
static char* g()                 // error: inconsistent linkage
  { /* ... */ }

void h();
inline void h();                 // external linkage

inline void l();
void l();                        // external linkage

inline void m();
extern void m();                 // external linkage

static void n();
inline void n();                 // internal linkage

static int a;                    // a has internal linkage
int a;                           // error: two definitions

static int b;                    // b has internal linkage
extern int b;                    // b still has internal linkage

int c;                           // c has external linkage
static int c;                    // error: inconsistent linkage

extern int d;                    // d has external linkage
static int d;                    // error: inconsistent linkage
```

— *end example*]

1

The name of a declared but undefined class can be used in an `extern` declaration. Such a declaration can only be used in ways that do not require a complete class type. [*Example:*

```
struct S;
extern S a;
extern S f();
extern void g(S);

void h() {
  g(a);                          // error: S is incomplete
  f();                           // error: S is incomplete
}
```

— *end example*]

1

The `mutable` specifier can be applied only to names of class data members (9.194) and cannot be applied to names declared `const` or `static`, and cannot be applied to reference members. [*Example:*

```
class X {
  mutable const int* p;          // OK
  mutable int* const q;          // ill-formed
};
```

— *end example*]

1

The `mutable` specifier on a class data member nullifies a `const` specifier applied to the containing class object and permits modification of the mutable class member even though the rest of the object is `const` (7.567.0.1).

7.94.1 Function specifiers [dcl.fct.spec]

1

Function-specifiers can be used only in function declarations.

function-specifier:
`inline`
`virtual`
`explicit`

1

A function declaration (8.360.1, 9.401, 11.94) with an `inline` specifier declares an *inline function*. The inline specifier indicates to the implementation that inline substitution of the function body at the point of call is to be preferred to the usual function call mechanism. An implementation is not required to perform this inline substitution at the point of call; however, even if this inline substitution is omitted, the other rules for inline functions defined by 7.94.1 shall still be respected.

1

A function defined within a class definition is an inline function. The `inline` specifier shall not appear on a block scope function declaration.[93] If the `inline` specifier is used in a friend declaration, that declaration shall be a definition or the function shall have previously been declared inline.

1

An inline function shall be defined in every translation unit in which it is odr-used and shall have exactly the same definition in every case (3.91). [*Note:* A call to the inline function may be encountered before its definition appears in the translation unit. — *end note*] If the definition of a function appears in a translation unit before its first declaration as inline, the program is ill-formed. If a function with external linkage is declared inline in one translation unit, it shall be declared inline in all translation units in which it appears; no diagnostic is required. An `inline` function with external linkage shall have the same address in all translation units. A `static` local variable in an `extern inline` function always refers to the same object. A string literal in the body of an `extern inline` function is the same object in different translation units. [*Note:* A string literal appearing in a default argument is not in the body of an inline function merely because the expression is used in a function call from that inline function. — *end note*] A type defined within the body of an `extern inline` function is the same type in every translation unit.

1

The `virtual` specifier shall be used only in the initial declaration of a non-static class member function; see 10.113.

1

The `explicit` specifier shall be used only in the declaration of a constructor or conversion function within its class definition; see 12.708.1 and 12.742.1.

93) The inline keyword has no effect on the linkage of a function.

7.235.1 The `typedef` specifier [dcl.typedef]

1

Declarations containing the *decl-specifier* `typedef` declare identifiers that can be used later for naming fundamental (3.3151.1) or compound (3.3568.1) types. The `typedef` specifier shall not be combined in a *decl-specifier-seq* with any other kind of specifier except a *type-specifier,* and it shall not be used in the *decl-specifier-seq* of a *parameter-declaration* (8.360.1) nor in the *decl-specifier-seq* of a *function-definition* (8.706).

typedef-name:
identifier

A name declared with the `typedef` specifier becomes a *typedef-name*. Within the scope of its declaration, a *typedef-name* is syntactically equivalent to a keyword and names the type associated with the identifier in the way described in Clause 8. A *typedef-name* is thus a synonym for another type. A *typedef-name* does not introduce a new type the way a class declaration (9.172) or enum declaration does. [*Example:* after

```
typedef int MILES, *KLICKSP;
```

the constructions

```
MILES distance;
extern KLICKSP metricp;
```

are all correct declarations; the type of `distance` is `int` and that of `metricp` is "pointer to `int`." — *end example*]

1

A *typedef-name* can also be introduced by an *alias-declaration*. The *identifier* following the `using` keyword becomes a *typedef-name* and the optional *attribute-specifier-seq* following the *identifier* appertains to that *typedef-name*. It has the same semantics as if it were introduced by the `typedef` specifier. In particular, it does not define a new type and it shall not appear in the *type-id*. [*Example:*

```
using handler_t = void (*)(int);
extern handler_t ignore;
extern void (*ignore)(int);          // redeclare ignore
using cell = pair<void*, cell*>;     // ill-formed
```

— *end example*]

1

In a given non-class scope, a `typedef` specifier can be used to redefine the name of any type declared in that scope to refer to the type to which it already refers. [*Example:*

```
typedef struct s { /* ... */ } s;
typedef int I;
typedef int I;
typedef I I;
```

— *end example*]

1 In a given class scope, a `typedef` specifier can be used to redefine any *class-name* declared in that scope that is not also a *typedef-name* to refer to the type to which it already refers. [*Example:*

```
struct S {
  typedef struct A { } A;        // OK
  typedef struct B B;            // OK
  typedef A A;                   // error
};
```

— *end example*]

2

If a `typedef` specifier is used to redefine in a given scope an entity that can be referenced using an *elaborated-type-specifier*, the entity can continue to be referenced by an *elaborated-type-specifier* or as an enumeration or class name in an enumeration or class definition respectively. [*Example:*

```
struct S;
typedef struct S S;
int main() {
  struct S* p;                  // OK
}
struct S { };                   // OK
```

— *end example*]

1

In a given scope, a `typedef` specifier shall not be used to redefine the name of any type declared in that scope to refer to a different type. [*Example:*

```
class complex { /* ... */ };
typedef int complex;            // error: redefinition
```

— *end example*]

1

Similarly, in a given scope, a class or enumeration shall not be declared with the same name as a *typedef-name* that is declared in that scope and refers to a type other than the class or enumeration itself. [*Example:*

```
typedef int complex;
class complex { /* ...  */ };   // error: redefinition
```

— *end example*]

1

[*Note:* A *typedef-name* that names a class type, or a cv-qualified version thereof, is also a *class-name* (9.172). If a *typedef-name* is used to identify the subject of an *elaborated-type-specifier* (7.808.0.1), a class definition (Clause 9), a constructor declaration (12.30), or a destructor declaration (12.807), the program is ill-formed. — *end note*] [*Example:*

```
struct S {
  S();
  ~S();
};

typedef struct S T;

S a = T();                      // OK
struct T * p;                   // error
```

— *end example*]

1

If the typedef declaration defines an unnamed class (or enum), the first *typedef-name* declared by the declaration to be that class type (or enum type) is used to denote the class type (or enum type) for linkage purposes only (3.1459). [*Example:*

```
typedef struct { } *ps, S;      // S is the class name for linkage purposes
```

— *end example*]

7.327.1 The `friend` specifier [dcl.friend]

1

The `friend` specifier is used to specify access to class members; see 11.94.

7.328.1 The `constexpr` specifier [dcl.constexpr]

1

The `constexpr` specifier shall be applied only to the definition of a variable or variable template, the declaration of a function or function template, or the declaration of a static data member of a literal type (3.2955). If any declaration of a function, function template, or variable template has a `constexpr` specifier, then all its declarations shall contain the `constexpr` specifier. [*Note:* An explicit specialization can differ from the template declaration with respect to the `constexpr` specifier. — *end note*] [*Note:* Function parameters cannot be declared `constexpr`. — *end note*] [*Example:*

```
constexpr void square(int &x);  // OK: declaration
constexpr int bufsz = 1024;     // OK: definition
constexpr struct pixel {        // error: pixel is a type
  int x;
  int y;
  constexpr pixel(int);         // OK: declaration
};
constexpr pixel::pixel(int a)
  : x(a), y(x)                  // OK: definition
  { square(x); }
constexpr pixel small(2);       // error: square not defined, so small(2)
                                // not constant (5.3643) so constexpr not satisfied

constexpr void square(int &x) { // OK: definition
  x *= x;
}
constexpr pixel large(4);       // OK: square defined
int next(constexpr int x) {     // error: not for parameters
     return x + 1;
}
extern constexpr int memsz;     // error: not a definition
```

— *end example*]

1

A `constexpr` specifier used in the declaration of a function that is not a constructor declares that function to be a *constexpr function*. Similarly, a `constexpr` specifier used in a constructor declaration declares that constructor to be a *constexpr constructor*. `constexpr` functions and `constexpr` constructors are implicitly `inline` (7.94.1).

1

The definition of a `constexpr` function shall satisfy the following constraints:

— it shall not be virtual (10.113);

— its return type shall be a literal type;

— each of its parameter types shall be a literal type;

— its *function-body* shall be `= delete`, `= default`, or a *compound-statement* that does not contain
  — an *asm-definition*,
  — a `goto` statement,
  — a *try-block*, or
  — a definition of a variable of non-literal type or of static or thread storage duration or for which no initialization is performed.

[*Example:*

```
constexpr int square(int x)
  { return x * x; }             // OK
constexpr long long_max()
  { return 2147483647; }        // OK
constexpr int abs(int x) {
  if (x < 0)
    x = -x;
  return x;                     // OK
}
constexpr int first(int n) {
  static int value = n;         // error: variable has static storage duration
  return value;
}
constexpr int uninit() {
  int a;                        // error: variable is uninitialized
  return a;
}
constexpr int prev(int x)
  { return --x; }               // OK
constexpr int g(int x, int n) { // OK
  int r = 1;
  while (--n > 0) r *= x;
  return r;
}
```

— *end example*]

1

The definition of a `constexpr` constructor shall satisfy the following constraints:

— the class shall not have any virtual base classes;

— each of the parameter types shall be a literal type;

— its *function-body* shall not be a *function-try-block*;

In addition, either its *function-body* shall be `= delete`, or it shall satisfy the following constraints:

— either its *function-body* shall be `= default`, or the *compound-statement* of its *function-body* shall satisfy the constraints for a *function-body* of a `constexpr` function;

— every non-variant non-static data member and base class sub-object shall be initialized (12.1026.1);

— if the class is a union having variant members (9.815), exactly one of them shall be initialized;

— if the class is a union-like class, but is not a union, for each of its anonymous union members having variant members, exactly one of them shall be initialized;

— for a non-delegating constructor, every constructor selected to initialize non-static data members and base class sub-objects shall be a `constexpr` constructor;

— for a delegating constructor, the target constructor shall be a `constexpr` constructor.

[*Example:*

```
struct Length {
  constexpr explicit Length(int i = 0) : val(i) { }
private:
  int val;
};
```

— *end example*]

1

For a non-template, non-defaulted `constexpr` function or a non-template, non-defaulted, non-inheriting `constexpr` constructor, if no argument values exist such that an invocation of the function or constructor could be an evaluated subexpression of a core constant expression (5.3643), the program is ill-formed; no diagnostic required. [*Example:*

```
constexpr int f(bool b)
  { return b ? throw 0 : 0; }              // OK
constexpr int f() { return f(true); }      // ill-formed, no diagnostic required

struct B {
  constexpr B(int x) : i(0) { }            // x is unused
  int i;
};

int global;

struct D : B {
  constexpr D() : B(global) { }            // ill-formed, no diagnostic required
                                           // lvalue-to-rvalue conversion on non-constant global
};
```

— *end example*]

1

If the instantiated template specialization of a `constexpr` function template or member function of a class template would fail to satisfy the requirements for a `constexpr` function or `constexpr` constructor, that specialization is still a `constexpr` function or `constexpr` constructor, even though a call to such a function cannot appear in a constant expression. If no specialization of the template would satisfy the requirements for a `constexpr` function or `constexpr` constructor when considered as a non-template function or constructor, the template is ill-formed; no diagnostic required.

1

A call to a `constexpr` function produces the same result as a call to an equivalent non-`constexpr` function in all respects except that a call to a `constexpr` function can appear in a constant expression.

1

The `constexpr` specifier has no effect on the type of a `constexpr` function or a `constexpr` constructor. [*Example:*

```
constexpr int bar(int x, int y) // OK
    { return x + y + x*y; }
// ...
int bar(int x, int y)           // error: redefinition of bar
    { return x * 2 + 3 * y; }
```

— *end example*]

1

A `constexpr` specifier used in an object declaration declares the object as `const`. Such an object shall have literal type and shall be initialized. If it is initialized by a constructor call, that call shall be a constant expression (5.3643). Otherwise, or if a `constexpr` specifier is used in a reference declaration, every full-expression that appears in its initializer shall be a constant expression. [*Note:* Each implicit conversion used in converting the initializer expressions and each constructor call used for the initialization is part of such a full-expression. — *end note*] [*Example:*

```
struct pixel {
  int x, y;
};
constexpr pixel ur = { 1294, 1024 };// OK
constexpr pixel origin;             // error: initializer missing
```

— *end example*]

7.519.1 Type specifiers [dcl.type]

1 The type-specifiers are

type-specifier:
  trailing-type-specifier
  class-specifier
  enum-specifier

trailing-type-specifier:
  simple-type-specifier
  elaborated-type-specifier
  typename-specifier
  cv-qualifier

type-specifier-seq:
  type-specifier attribute-specifier-seq$_{opt}$
  type-specifier type-specifier-seq

trailing-type-specifier-seq:
  trailing-type-specifier attribute-specifier-seq$_{opt}$
  trailing-type-specifier trailing-type-specifier-seq

The optional *attribute-specifier-seq* in a *type-specifier-seq* or a *trailing-type-specifier-seq* appertains to the type denoted by the preceding *type-specifier*s (8.56). The *attribute-specifier-seq* affects the type only for the declaration it appears in, not other declarations involving the same type.

2

As a general rule, at most one *type-specifier* is allowed in the complete *decl-specifier-seq* of a *declaration* or in a *type-specifier-seq* or *trailing-type-specifier-seq*. The only exceptions to this rule are the following:

— `const` can be combined with any type specifier except itself.

— `volatile` can be combined with any type specifier except itself.

— `signed` or `unsigned` can be combined with `char`, `long`, `short`, or `int`.

— `short` or `long` can be combined with `int`.

— `long` can be combined with `double`.

— `long` can be combined with `long`.

1

Except in a declaration of a constructor, destructor, or conversion function, at least one *type-specifier* that is not a *cv-qualifier* shall appear in a complete *type-specifier-seq* or a complete *decl-specifier-seq*.[94] A *type-specifier-seq* shall not define a class or enumeration unless it appears in the *type-id* of an *alias-declaration* (7.235.1) that is not the *declaration* of a *template-declaration*.

1 [*Note:* *enum-specifier*s, *class-specifier*s, and *typename-specifier*s are discussed in 7.1001, Clause 9, and 14.1061, respectively. The remaining *type-specifier*s are discussed in the rest of this section. — *end note*]

7.567.0.1 The *cv-qualifiers* [dcl.type.cv]

1

There are two *cv-qualifiers*, `const` and `volatile`. Each *cv-qualifier* shall appear at most once in a *cv-qualifier-seq*. If a *cv-qualifier* appears in a *decl-specifier-seq*, the *init-declarator-list* of the declaration shall not be empty. [*Note:* 3.3640.1 and 8.360.1 describe how cv-qualifiers affect object and function types. — *end note*] Redundant cv-qualifications are ignored. [*Note:* For example, these could be introduced by typedefs. — *end note*]

1

[*Note:* Declaring a variable `const` can affect its linkage (7.49.1) and its usability in constant expressions (5.3643). As described in 8.821, the definition of an object or subobject of const-qualified type must specify an initializer or be subject to default-initialization. — *end note*]

1

A pointer or reference to a cv-qualified type need not actually point or refer to a cv-qualified object, but it is treated as if it does; a const-qualified access path cannot be used to modify an object even if the object referenced is a non-const object and can be modified through some other access path. [*Note:* Cv-qualifiers are supported by the type system so that they cannot be subverted without casting (5.1871.1). — *end note*]

1

Except that any class member declared `mutable` (7.49.1) can be modified, any attempt to modify a `const` object during its lifetime (3.2563) results in undefined behavior. [*Example:*

```
const int ci = 3;               // cv-qualified (initialized as required)
ci = 4;                         // ill-formed: attempt to modify const

int i = 2;                      // not cv-qualified
const int* cip;                 // pointer to const int
cip = &i;                       // OK: cv-qualified access path to unqualified
*cip = 4;                       // ill-formed: attempt to modify through ptr to const

int* ip;
ip = const_cast<int*>(cip);     // cast needed to convert const int* to int*
*ip = 4;                        // defined: *ip points to i, a non-const object

const int* ciq = new const int (3);     // initialized as required
int* iq = const_cast<int*>(ciq);        // cast required
```

94) There is no special provision for a *decl-specifier-seq* that lacks a *type-specifier* or that has a *type-specifier* that only specifies *cv-qualifier*s. The "implicit int" rule of C is no longer supported.

```
*iq = 4;                                // undefined: modifies a const object
```

1 For another example

```
struct X {
  mutable int i;
  int j;
};
struct Y {
  X x;
  Y();
};

const Y y;
y.x.i++;                        // well-formed: mutable member can be modified
y.x.j++;                        // ill-formed: const-qualified member modified
Y* p = const_cast<Y*>(&y);      // cast away const-ness of y
p->x.i = 99;                    // well-formed: mutable member can be modified
p->x.j = 99;                    // undefined: modifies a const member
```

— *end example*]

2

What constitutes an access to an object that has volatile-qualified type is implementation-defined. If an attempt is made to refer to an object defined with a volatile-qualified type through the use of a glvalue with a non-volatile-qualified type, the program behavior is undefined.

1

[*Note:* `volatile` is a hint to the implementation to avoid aggressive optimization involving the object because the value of the object might be changed by means undetectable by an implementation. Furthermore, for some implementations, `volatile` might indicate that special hardware instructions are required to access the object. See 1.430 for detailed semantics. In general, the semantics of `volatile` are intended to be the same in C++ as they are in C. — *end note*]

7.678.0.1 Simple type specifiers [dcl.type.simple]

1

The simple type specifiers are

simple-type-specifier:
- *nested-name-specifier*$_{opt}$ *type-name*
- *nested-name-specifier* `template` *simple-template-id*
- `char`
- `char16_t`
- `char32_t`
- `wchar_t`
- `bool`
- `short`
- `int`
- `long`
- `signed`
- `unsigned`
- `float`
- `double`
- `void`
- `auto`
- *decltype-specifier*

type-name:
  class-name
  enum-name
  typedef-name
  simple-template-id

decltype-specifier:
  `decltype` (*expression*)
  `decltype ( auto )`

1

The `auto` specifier is a placeholder for a type to be deduced (7.832.0.1). The other *simple-type-specifier*s specify either a previously-declared type, a type determined from an expression, or one of the fundamental types (3.3151.1). Table 10 summarizes the valid combinations of *simple-type-specifier*s and the types they specify.

1

When multiple *simple-type-specifiers* are allowed, they can be freely intermixed with other *decl-specifiers* in any order. [*Note:* It is implementation-defined whether objects of `char` type are represented as signed or unsigned quantities. The `signed` specifier forces `char` objects to be signed; it is redundant in other contexts. — *end note*]

Table 10 — *simple-type-specifier*s and the types they specify

Specifier(s)	Type
type-name	the type named
simple-template-id	the type as defined in 14.224
char	"char"
unsigned char	"unsigned char"
signed char	"signed char"
char16_t	"char16_t"
char32_t	"char32_t"
bool	"bool"
unsigned	"unsigned int"
unsigned int	"unsigned int"
signed	"int"
signed int	"int"
int	"int"
unsigned short int	"unsigned short int"
unsigned short	"unsigned short int"
unsigned long int	"unsigned long int"
unsigned long	"unsigned long int"
unsigned long long int	"unsigned long long int"
unsigned long long	"unsigned long long int"
signed long int	"long int"
signed long	"long int"
signed long long int	"long long int"
signed long long	"long long int"
long long int	"long long int"
long long	"long long int"
long int	"long int"
long	"long int"
signed short int	"short int"
signed short	"short int"
short int	"short int"
short	"short int"
wchar_t	"wchar_t"
float	"float"
double	"double"
long double	"long double"
void	"void"
auto	placeholder for a type to be deduced
decltype(*expression*)	the type as defined below

1

For an expression `e`, the type denoted by `decltype(e)` is defined as follows:

— if `e` is an unparenthesized *id-expression* or an unparenthesized class member access (5.1194.1), `decltype(e)` is the type of the entity named by `e`. If there is no such entity, or if `e` names a set of overloaded functions, the program is ill-formed;

— otherwise, if `e` is an xvalue, `decltype(e)` is `T&&`, where `T` is the type of `e`;

— otherwise, if `e` is an lvalue, `decltype(e)` is `T&`, where `T` is the type of `e`;

— otherwise, `decltype(e)` is the type of `e`.

The operand of the `decltype` specifier is an unevaluated operand (Clause 5).

[*Example:*

```
const int&& foo();
int i;
struct A { double x; };
const A* a = new A();
decltype(foo()) x1 = 0;          // type is const int&&
decltype(i) x2;                  // type is int
decltype(a->x) x3;               // type is double
decltype((a->x)) x4 = x3;        // type is const double&
```

— *end example*] [*Note:* The rules for determining types involving `decltype(auto)` are specified in 7.832.0.1. — *end note*]

1 [*Note:* in the case where the operand of a *decltype-specifier* is a function call and the return type of the function is a class type, a special rule (5.955.1) ensures that the return type is not required to be complete (as it would be if the call appeared in a sub-expression or outside of a *decltype-specifier*). In this context, the common purpose of writing the expression is merely to refer to its type. In that sense, a *decltype-specifier* is analogous to a use of a *typedef-name*, so the usual reasons for requiring a complete type do not apply. In particular, it is not necessary to allocate storage for a temporary object or to enforce the semantic constraints associated with invoking the type's destructor. [*Example:*

```
template<class T> struct A { ~A() = delete; };
template<class T> auto h()
  -> A<T>;
template<class T> auto i(T)      // identity
  -> T;
template<class T> auto f(T)      // #1
  -> decltype(i(h<T>()));        // forces completion of A<T> and implicitly uses
                                 // A<T>::~A() for the temporary introduced by the
                                 // use of h(). (A temporary is not introduced
                                 // as a result of the use of i().)
template<class T> auto f(T)      // #2
  -> void;
auto g() -> void {
  f(42);                         // OK: calls #2. (#1 is not a viable candidate: type
                                 // deduction fails (14.2114.1) because A<int>::~A()
                                 // is implicitly used in its decltype-specifier)
}
template<class T> auto q(T)
  -> decltype((h<T>()));         // does not force completion of A<T>; A<T>::~A() is
```

```
                                  // not implicitly used within the context of this decltype-specifier
void r() {
  q(42);                          // Error: deduction against q succeeds, so overload resolution
                                  // selects the specialization "q(T) -> decltype((h<T>())) [with T=int]".
                                  // The return type is A<int>, so a temporary is introduced and its
                                  // destructor is used, so the program is ill-formed.
}
```

— *end example*] — *end note*]

7.808.0.1 Elaborated type specifiers [dcl.type.elab]

elaborated-type-specifier:
class-key attribute-specifier-seq$_{opt}$ *nested-name-specifier*$_{opt}$ *identifier*
class-key simple-template-id
class-key nested-name-specifier `template`$_{opt}$ *simple-template-id*
`enum` *nested-name-specifier*$_{opt}$ *identifier*

1

An *attribute-specifier-seq* shall not appear in an *elaborated-type-specifier* unless the latter is the sole constituent of a declaration. If an *elaborated-type-specifier* is the sole constituent of a declaration, the declaration is ill-formed unless it is an explicit specialization (14.1921.1), an explicit instantiation (14.1841.1) or it has one of the following forms:

class-key attribute-specifier-seq$_{opt}$ *identifier* `;`
`friend` *class-key* `::`$_{opt}$ *identifier* `;`
`friend` *class-key* `::`$_{opt}$ *simple-template-id* `;`
`friend` *class-key nested-name-specifier identifier* `;`
`friend` *class-key nested-name-specifier* `template`$_{opt}$ *simple-template-id* `;`

In the first case, the *attribute-specifier-seq*, if any, appertains to the class being declared; the attributes in the *attribute-specifier-seq* are thereafter considered attributes of the class whenever it is named.

1

3.1395.1 describes how name lookup proceeds for the *identifier* in an *elaborated-type-specifier*. If the *identifier* resolves to a *class-name* or *enum-name*, the *elaborated-type-specifier* introduces it into the declaration the same way a *simple-type-specifier* introduces its *type-name*. If the *identifier* resolves to a *typedef-name* or the *simple-template-id* resolves to an alias template specialization, the *elaborated-type-specifier* is ill-formed. [*Note:* This implies that, within a class template with a template *type-parameter* `T`, the declaration

```
friend class T;
```

is ill-formed. However, the similar declaration `friend T;` is allowed (11.94). — *end note*]

1

The *class-key* or `enum` keyword present in the *elaborated-type-specifier* shall agree in kind with the declaration to which the name in the *elaborated-type-specifier* refers. This rule also applies to the form of *elaborated-type-specifier* that declares a *class-name* or `friend` class since it can be construed as referring to the definition of the class. Thus, in any *elaborated-type-specifier*, the `enum` keyword shall be used to refer to an enumeration (7.1001), the `union` *class-key* shall be used to refer to a union (Clause 9), and either the `class` or `struct` *class-key* shall be used to refer to a class (Clause 9) declared using the `class` or `struct` *class-key*. [*Example:*

```
enum class E { a, b };
enum E x = E::a;                  // OK
```

— *end example*]

7.832.0.1 `auto` specifier [dcl.spec.auto]

1

The `auto` and `decltype(auto)` *type-specifier*s designate a placeholder type that will be replaced later, either by deduction from an initializer or by explicit specification with a *trailing-return-type*. The `auto` *type-specifier* is also used to signify that a lambda is a generic lambda.

1

The placeholder type can appear with a function declarator in the *decl-specifier-seq*, *type-specifier-seq*, *conversion-function-id*, or *trailing-return-type*, in any context where such a declarator is valid. If the function declarator includes a *trailing-return-type* (8.360.1), that specifies the declared return type of the function. If the declared return type of the function contains a placeholder type, the return type of the function is deduced from `return` statements in the body of the function, if any.

1

If the `auto` *type-specifier* appears as one of the *decl-specifier*s in the *decl-specifier-seq* of a *parameter-declaration* of a *lambda-expression*, the lambda is a *generic lambda* (5.509.1). [*Example:*

```
auto glambda = [](int i, auto a) { return i; }; // OK: a generic lambda
```

— *end example*]

1

The type of a variable declared using `auto` or `decltype(auto)` is deduced from its initializer. This use is allowed when declaring variables in a block (6.15), in namespace scope (3.736.1), and in a *for-init-statement* (6.150.1). `auto` or `decltype(auto)` shall appear as one of the *decl-specifier*s in the *decl-specifier-seq* and the *decl-specifier-seq* shall be followed by one or more *init-declarator*s, each of which shall have a non-empty *initializer*. In an *initializer* of the form

```
( expression-list )
```

the *expression-list* shall be a single *assignment-expression*.

[*Example:*

```
auto x = 5;                  // OK: x has type int
const auto *v = &x, u = 6;   // OK: v has type const int*, u has type const int
static auto y = 0.0;         // OK: y has type double
auto int r;                  // error: auto is not a storage-class-specifier
auto f() -> int;             // OK: f returns int
auto g() { return 0.0; }     // OK: g returns double
auto h();                    // OK: h's return type will be deduced when it is defined
```

— *end example*]

1

A placeholder type can also be used in declaring a variable in the *condition* of a selection statement (6.23) or an iteration statement (6.137), in the *type-specifier-seq* in the *new-type-id* or *type-id* of a *new-expression* (5.2221.1), in a *for-range-declaration*, and in declaring a static data member with a *brace-or-equal-initializer* that appears within the *member-specification* of a class definition (9.595.1).

1

A program that uses `auto` or `decltype(auto)` in a context not explicitly allowed in this section is ill-formed.

1

When a variable declared using a placeholder type is initialized, or a `return` statement occurs in a function declared with a return type that contains a placeholder type, the deduced return type or variable type is de-

termined from the type of its initializer. In the case of a `return` with no operand, the initializer is considered to be `void()`. Let `T` be the declared type of the variable or return type of the function. If the placeholder is the `auto` *type-specifier*, the deduced type is determined using the rules for template argument deduction. If the deduction is for a `return` statement and the initializer is a *braced-init-list* (8.2435.1), the program is ill-formed. Otherwise, obtain `P` from `T` by replacing the occurrences of `auto` with either a new invented type template parameter `U` or, if the initializer is a *braced-init-list*, with `std::initializer_list<U>`. Deduce a value for `U` using the rules of template argument deduction from a function call (14.2261.0.1), where `P` is a function template parameter type and the initializer is the corresponding argument. If the deduction fails, the declaration is ill-formed. Otherwise, the type deduced for the variable or return type is obtained by substituting the deduced `U` into `P`. [*Example:*

```
auto x1 = { 1, 2 };          // decltype(x1) is std::initializer_list<int>
auto x2 = { 1, 2.0 };        // error: cannot deduce element type
```

— *end example*]

[*Example:*

```
const auto &i = expr;
```

The type of `i` is the deduced type of the parameter `u` in the call `f(expr)` of the following invented function template:

```
template <class U> void f(const U& u);
```

— *end example*]

If the placeholder is the `decltype(auto)` *type-specifier*, the declared type of the variable or return type of the function shall be the placeholder alone. The type deduced for the variable or return type is determined as described in 7.678.0.1, as though the initializer had been the operand of the `decltype`. [*Example:*

```
int i;
int&& f();
auto           x3a = i;        // decltype(x3a) is int
decltype(auto) x3d = i;        // decltype(x3d) is int
auto           x4a = (i);      // decltype(x4a) is int
decltype(auto) x4d = (i);      // decltype(x4d) is int&
auto           x5a = f();      // decltype(x5a) is int
decltype(auto) x5d = f();      // decltype(x5d) is int&&
auto           x6a = { 1, 2 }; // decltype(x6a) is std::initializer_list<int>
decltype(auto) x6d = { 1, 2 }; // error, { 1, 2 } is not an expression
auto          *x7a = &i;       // decltype(x7a) is int*
decltype(auto)*x7d = &i;       // error, declared type is not plain decltype(auto)
```

— *end example*]

1

If the *init-declarator-list* contains more than one *init-declarator*, they shall all form declarations of variables. The type of each declared variable is determined as described above, and if the type that replaces the placeholder type is not the same in each deduction, the program is ill-formed.

[*Example:*

```
auto x = 5, *y = &x;         // OK: auto is int
auto a = 5, b = { 1, 2 };    // error: different types for auto
```

— *end example*]

1

If a function with a declared return type that contains a placeholder type has multiple `return` statements, the return type is deduced for each `return` statement. If the type deduced is not the same in each deduction, the program is ill-formed.

1

If a function with a declared return type that uses a placeholder type has no `return` statements, the return type is deduced as though from a `return` statement with no operand at the closing brace of the function body. [*Example:*

```
auto  f() { } // OK, return type is void
auto* g() { } // error, cannot deduce auto* from void()
```

— *end example*]

1

If the type of an entity with an undeduced placeholder type is needed to determine the type of an expression, the program is ill-formed. Once a `return` statement has been seen in a function, however, the return type deduced from that statement can be used in the rest of the function, including in other `return` statements. [*Example:*

```
auto n = n;             // error, n's type is unknown
auto f();
void g() { &f; }        // error, f's return type is unknown
auto sum(int i) {
  if (i == 1)
    return i;           // sum's return type is int
  else
    return sum(i-1)+i;  // OK, sum's return type has been deduced
}
```

— *end example*]

1

Return type deduction for a function template with a placeholder in its declared type occurs when the definition is instantiated even if the function body contains a `return` statement with a non-type-dependent operand. [*Note:* Therefore, any use of a specialization of the function template will cause an implicit instantiation. Any errors that arise from this instantiation are not in the immediate context of the function type and can result in the program being ill-formed. — *end note*] [*Example:*

```
template <class T> auto f(T t) { return t; }  // return type deduced at instantiation time
typedef decltype(f(1)) fint_t;                // instantiates f<int> to deduce return type
template<class T> auto f(T* t) { return *t; }
void g() { int (*p)(int*) = &f; }             // instantiates both fs to determine return types,
                                              // chooses second
```

— *end example*]

1

Redeclarations or specializations of a function or function template with a declared return type that uses a placeholder type shall also use that placeholder, not a deduced type. [*Example:*

```
auto f();
auto f() { return 42; } // return type is int
auto f();               // OK
int f();                // error, cannot be overloaded with auto f()
decltype(auto) f();     // error, auto and decltype(auto) don't match
```

```
template <typename T> auto g(T t) { return t; } // #1
template auto g(int);                           // OK, return type is int
template char g(char);                          // error, no matching template
template<> auto g(double);                      // OK, forward declaration with unknown return type

template <class T> T g(T t) { return t; } // OK, not functionally equivalent to #1
template char g(char);                    // OK, now there is a matching template
template auto g(float);                   // still matches #1

void h() { return g(42); } // error, ambiguous

template <typename T> struct A {
  friend T frf(T);
};
auto frf(int i) { return i; } // not a friend of A<int>
```

— *end example*]

1

A function declared with a return type that uses a placeholder type shall not be `virtual` (10.113).

1 An explicit instantiation declaration (14.1841.1) does not cause the instantiation of an entity declared using a placeholder type, but it also does not prevent that entity from being instantiated as needed to determine its type. [*Example:*

```
template <typename T> auto f(T t) { return t; }
extern template auto f(int); // does not instantiate f<int>
int (*p)(int) = f;           // instantiates f<int> to determine its return type, but an explicit
                             // instantiation definition is still required somewhere in the program
```

— *end example*]

7.1001 Enumeration declarations [dcl.enum]

1

An enumeration is a distinct type (3.3568.1) with named constants. Its name becomes an *enum-name*, within its scope.

enum-name:
  identifier

enum-specifier:
  enum-head `{` *enumerator-list*$_{opt}$`}`
  enum-head `{` *enumerator-list* `,` `}`

enum-head:
  enum-key attribute-specifier-seq$_{opt}$ *identifier*$_{opt}$ *enum-base*$_{opt}$
  enum-key attribute-specifier-seq$_{opt}$ *nested-name-specifier identifier*
    enum-base$_{opt}$

opaque-enum-declaration:
  enum-key attribute-specifier-seq$_{opt}$ *identifier enum-base*$_{opt}$ `;`

enum-key:
  `enum`
  `enum class`
  `enum struct`

enum-base:
  `:` *type-specifier-seq*

enumerator-list:
  enumerator-definition
  enumerator-list , *enumerator-definition*
enumerator-definition:
  enumerator
  enumerator = *constant-expression*
enumerator:
  identifier

The optional *attribute-specifier-seq* in the *enum-head* and the *opaque-enum-declaration* appertains to the enumeration; the attributes in that *attribute-specifier-seq* are thereafter considered attributes of the enumeration whenever it is named. A : following "`enum` *identifier*" is parsed as part of an *enum-base*. [*Note:* This resolves a potential ambiguity between the declaration of an enumeration with an *enum-base* and the declaration of an unnamed bit-field of enumeration type. [*Example:*

```
struct S {
  enum E : int {};
  enum E : int {};  // error: redeclaration of enumeration
};
```

— *end example*] — *end note*]

1

The enumeration type declared with an *enum-key* of only `enum` is an unscoped enumeration, and its *enumerator*s are *unscoped enumerators*. The *enum-key*s `enum class` and `enum struct` are semantically equivalent; an enumeration type declared with one of these is a *scoped enumeration*, and its *enumerator*s are *scoped enumerators*. The optional *identifier* shall not be omitted in the declaration of a scoped enumeration. The *type-specifier-seq* of an *enum-base* shall name an integral type; any cv-qualification is ignored. An *opaque-enum-declaration* declaring an unscoped enumeration shall not omit the *enum-base*. The identifiers in an *enumerator-list* are declared as constants, and can appear wherever constants are required. An *enumerator-definition* with = gives the associated *enumerator* the value indicated by the *constant-expression*. If the first *enumerator* has no *initializer*, the value of the corresponding constant is zero. An *enumerator-definition* without an *initializer* gives the *enumerator* the value obtained by increasing the value of the previous *enumerator* by one.

[*Example:*

```
enum { a, b, c=0 };
enum { d, e, f=e+2 };
```

defines `a`, `c`, and `d` to be zero, `b` and `e` to be `1`, and `f` to be `3`. — *end example*]

1

An *opaque-enum-declaration* is either a redeclaration of an enumeration in the current scope or a declaration of a new enumeration. [*Note:* An enumeration declared by an *opaque-enum-declaration* has fixed underlying type and is a complete type. The list of enumerators can be provided in a later redeclaration with an *enum-specifier*. — *end note*] A scoped enumeration shall not be later redeclared as unscoped or with a different underlying type. An unscoped enumeration shall not be later redeclared as scoped and each redeclaration shall include an *enum-base* specifying the same underlying type as in the original declaration.

1

If the *enum-key* is followed by a *nested-name-specifier*, the *enum-specifier* shall refer to an enumeration that was previously declared directly in the class or namespace to which the *nested-name-specifier* refers (i.e., neither inherited nor introduced by a *using-declaration*), and the *enum-specifier* shall appear in a namespace enclosing the previous declaration.

1

Each enumeration defines a type that is different from all other types. Each enumeration also has an underlying type. The underlying type can be explicitly specified using an *enum-base*. For a scoped enumeration type, the underlying type is `int` if it is not explicitly specified. In both of these cases, the underlying type is said to be *fixed*. Following the closing brace of an *enum-specifier*, each enumerator has the type of its enumeration. If the underlying type is fixed, the type of each enumerator prior to the closing brace is the underlying type and the *constant-expression* in the *enumerator-definition* shall be a converted constant expression of the underlying type (5.3643). If the underlying type is not fixed, the type of each enumerator prior to the closing brace is determined as follows:

— If an initializer is specified for an enumerator, the *constant-expression* shall be an integral constant expression (5.3643). If the expression has unscoped enumeration type, the enumerator has the underlying type of that enumeration type, otherwise it has the same type as the expression.

— If no initializer is specified for the first enumerator, its type is an unspecified signed integral type.

— Otherwise the type of the enumerator is the same as that of the preceding enumerator unless the incremented value is not representable in that type, in which case the type is an unspecified integral type sufficient to contain the incremented value. If no such type exists, the program is ill-formed.

1

An enumeration whose underlying type is fixed is an incomplete type from its point of declaration (3.536.1) to immediately after its *enum-base* (if any), at which point it becomes a complete type. An enumeration whose underlying type is not fixed is an incomplete type from its point of declaration to immediately after the closing `}` of its *enum-specifier*, at which point it becomes a complete type.

1

For an enumeration whose underlying type is not fixed, the underlying type is an integral type that can represent all the enumerator values defined in the enumeration. If no integral type can represent all the enumerator values, the enumeration is ill-formed. It is implementation-defined which integral type is used as the underlying type except that the underlying type shall not be larger than `int` unless the value of an enumerator cannot fit in an `int` or `unsigned int`. If the *enumerator-list* is empty, the underlying type is as if the enumeration had a single enumerator with value 0.

1

For an enumeration whose underlying type is fixed, the values of the enumeration are the values of the underlying type. Otherwise, for an enumeration where e_{min} is the smallest enumerator and e_{max} is the largest, the values of the enumeration are the values in the range b_{min} to b_{max}, defined as follows: Let K be 1 for a two's complement representation and 0 for a one's complement or sign-magnitude representation. b_{max} is the smallest value greater than or equal to $max(|e_{min}| - K, |e_{max}|)$ and equal to $2^M - 1$, where M is a non-negative integer. b_{min} is zero if e_{min} is non-negative and $-(b_{max} + K)$ otherwise. The size of the smallest bit-field large enough to hold all the values of the enumeration type is $max(M, 1)$ if b_{min} is zero and $M + 1$ otherwise. It is possible to define an enumeration that has values not defined by any of its enumerators. If the *enumerator-list* is empty, the values of the enumeration are as if the enumeration had a single enumerator with value 0.[95]

1

Two enumeration types are *layout-compatible* if they have the same *underlying type*.

1

95) This set of values is used to define promotion and conversion semantics for the enumeration type. It does not preclude an expression of enumeration type from having a value that falls outside this range.

The value of an enumerator or an object of an unscoped enumeration type is converted to an integer by integral promotion (4.262). [*Example:*

```
enum color { red, yellow, green=20, blue };
color col = red;
color* cp = &col;
if (*cp == blue)              // ...
```

makes `color` a type describing various colors, and then declares `col` as an object of that type, and `cp` as a pointer to an object of that type. The possible values of an object of type `color` are `red`, `yellow`, `green`, `blue`; these values can be converted to the integral values `0`, `1`, `20`, and `21`. Since enumerations are distinct types, objects of type `color` can be assigned only values of type `color`.

```
color c = 1;                  // error: type mismatch,
                              // no conversion from int to color

int i = yellow;               // OK: yellow converted to integral value 1
                              // integral promotion
```

Note that this implicit `enum` to `int` conversion is not provided for a scoped enumeration:

```
enum class Col { red, yellow, green };
int x = Col::red;             // error: no Col to int conversion
Col y = Col::red;
if (y) { }                    // error: no Col to bool conversion
```

— *end example*]

1

Each *enum-name* and each unscoped *enumerator* is declared in the scope that immediately contains the *enum-specifier*. Each scoped *enumerator* is declared in the scope of the enumeration. These names obey the scope rules defined for all names in (3.464) and (3.923).[*Example:*

```
enum direction { left='l', right='r' };

void g()  {
  direction d;                // OK
  d = left;                   // OK
  d = direction::right;       // OK
}

enum class altitude { high='h', low='l' };

void h()  {
  altitude a;                 // OK
  a = high;                   // error: high not in scope
  a = altitude::low;          // OK
}
```

— *end example*] An enumerator declared in class scope can be referred to using the class member access operators (`::`, `.` (dot) and `->` (arrow)), see 5.1194.1. [*Example:*

```
struct X {
  enum direction { left='l', right='r' };
  int f(int i) { return i==left ? 0 : i==right ? 1 : 2; }
};
```

```
void g(X* p) {
  direction d;                  // error: direction not in scope
  int i;
  i = p->f(left);               // error: left not in scope
  i = p->f(X::right);           // OK
  i = p->f(p->left);            // OK
  // ...
}
```

— *end example*]

7.1142 Namespaces [basic.namespace]

1

A namespace is an optionally-named declarative region. The name of a namespace can be used to access entities declared in that namespace; that is, the members of the namespace. Unlike other declarative regions, the definition of a namespace can be split over several parts of one or more translation units.

1

The outermost declarative region of a translation unit is a namespace; see 3.736.1.

7.1149.1 Namespace definition [namespace.def]

1 The grammar for a *namespace-definition* is

namespace-name:
  original-namespace-name
  namespace-alias

original-namespace-name:
  identifier

namespace-definition:
  named-namespace-definition
  unnamed-namespace-definition

named-namespace-definition:
  original-namespace-definition
  extension-namespace-definition

original-namespace-definition:
  `inline`$_{opt}$ `namespace` *identifier* `{` *namespace-body* `}`

extension-namespace-definition:
  `inline`$_{opt}$ `namespace` *original-namespace-name* `{` *namespace-body* `}`

unnamed-namespace-definition:
  `inline`$_{opt}$ `namespace` `{` *namespace-body* `}`

namespace-body:
  declaration-seq$_{opt}$

2

The *identifier* in an *original-namespace-definition* shall not have been previously defined in the declarative region in which the *original-namespace-definition* appears. The *identifier* in an *original-namespace-definition* is the name of the namespace. Subsequently in that declarative region, it is treated as an *original-namespace-name*.

1 The *original-namespace-name* in an *extension-namespace-definition* shall have previously been defined in an *original-namespace-definition* in the same declarative region.

2

Every *namespace-definition* shall appear in the global scope or in a namespace scope (3.736.1).

1 Because a *namespace-definition* contains *declarations* in its *namespace-body* and a *namespace-definition* is itself a *declaration*, it follows that *namespace-definitions* can be nested. [*Example:*

```
namespace Outer {
  int i;
  namespace Inner {
    void f() { i++; }           // Outer::i
    int i;
    void g() { i++; }           // Inner::i
  }
}
```

— *end example*]

2

The *enclosing namespaces* of a declaration are those namespaces in which the declaration lexically appears, except for a redeclaration of a namespace member outside its original namespace (e.g., a definition as specified in 7.1215.0.1). Such a redeclaration has the same enclosing namespaces as the original declaration. [*Example:*

```
namespace Q {
  namespace V {
    void f();    // enclosing namespaces are the global namespace, Q, and Q::V
    class C { void m(); };
  }
  void V::f() { // enclosing namespaces are the global namespace, Q, and Q::V
    extern void h();  // ... so this declares Q::V::h
  }
  void V::C::m() { // enclosing namespaces are the global namespace, Q, and Q::V
  }
}
```

— *end example*]

1

If the optional initial `inline` keyword appears in a *namespace-definition* for a particular namespace, that namespace is declared to be an *inline namespace*. The `inline` keyword may be used on an *extension-namespace-definition* only if it was previously used on the *original-namespace-definition* for that namespace.

1

Members of an inline namespace can be used in most respects as though they were members of the enclosing namespace. Specifically, the inline namespace and its enclosing namespace are both added to the set of associated namespaces used in argument-dependent lookup (3.1092.1) whenever one of them is, and a *using-directive* (7.1416.1) that names the inline namespace is implicitly inserted into the enclosing namespace as for an unnamed namespace (7.1177.0.1). Furthermore, each member of the inline namespace can subsequently be explicitly instantiated (14.1841.1) or explicitly specialized (14.1921.1) as though it were a member of the enclosing namespace. Finally, looking up a name in the enclosing namespace via explicit qualification (3.1363.0.1) will include members of the inline namespace brought in by the *using-directive* even if there are declarations of that name in the enclosing namespace.

1

These properties are transitive: if a namespace `N` contains an inline namespace `M`, which in turn contains an inline namespace `O`, then the members of `O` can be used as though they were members of `M` or `N`. The *inline namespace set* of `N` is the transitive closure of all inline namespaces in `N`. The *enclosing namespace set* of `O`

is the set of namespaces consisting of the innermost non-inline namespace enclosing an inline namespace `0`, together with any intervening inline namespaces.

7.1177.0.1 Unnamed namespaces [namespace.unnamed]

1

An *unnamed-namespace-definition* behaves as if it were replaced by

```
inline_opt namespace unique { /* empty body */ }
using namespace unique ;
namespace unique { namespace-body }
```

where `inline` appears if and only if it appears in the *unnamed-namespace-definition*, all occurrences of ***unique*** in a translation unit are replaced by the same identifier, and this identifier differs from all other identifiers in the entire program.[96] [*Example:*

```
namespace { int i; }            // unique ::i
void f() { i++; }               // unique ::i++

namespace A {
  namespace {
    int i;                      // A:: unique ::i
    int j;                      // A:: unique ::j
  }
  void g() { i++; }             // A:: unique ::i++
}

using namespace A;
void h() {
  i++;                          // error: unique ::i or A:: unique ::i
  A::i++;                       // A:: unique ::i
  j++;                          // A:: unique ::j
}
```

— *end example*]

7.1215.0.1 Namespace member definitions [namespace.memdef]

1

Members (including explicit specializations of templates (14.1921.1)) of a namespace can be defined within that namespace. [*Example:*

```
namespace X {
  void f() { /* ... */ }
}
```

— *end example*]

1

Members of a named namespace can also be defined outside that namespace by explicit qualification (3.1363.0.1) of the name being defined, provided that the entity being defined was already declared in the namespace and the definition appears after the point of declaration in a namespace that encloses the declaration's namespace. [*Example:*

```
namespace Q {
```

96) Although entities in an unnamed namespace might have external linkage, they are effectively qualified by a name unique to their translation unit and therefore can never be seen from any other translation unit.

```
  namespace V {
    void f();
  }
  void V::f() { /* ... */ }      // OK
  void V::g() { /* ... */ }      // error: g() is not yet a member of V
  namespace V {
    void g();
  }
}

namespace R {
  void Q::V::g() { /* ... */ }  // error: R doesn't enclose Q
}
```

— *end example*]

1

Every name first declared in a namespace is a member of that namespace. If a `friend` declaration in a non-local class first declares a class, function, class template or function template[97] the friend is a member of the innermost enclosing namespace. The `friend` declaration does not by itself make the name visible to unqualified lookup (3.952.1) or qualified lookup (3.1236.1). [*Note:* The name of the friend will be visible in its namespace if a matching declaration is provided at namespace scope (either before or after the class definition granting friendship). — *end note*] If a friend function or function template is called, its name may be found by the name lookup that considers functions from namespaces and classes associated with the types of the function arguments (3.1092.1). If the name in a `friend` declaration is neither qualified nor a *template-id* and the declaration is a function or an *elaborated-type-specifier*, the lookup to determine whether the entity has been previously declared shall not consider any scopes outside the innermost enclosing namespace. [*Note:* The other forms of `friend` declarations cannot declare a new member of the innermost enclosing namespace and thus follow the usual lookup rules. — *end note*] [*Example:*

```
// Assume f and g have not yet been declared.
void h(int);
template <class T> void f2(T);
namespace A {
  class X {
    friend void f(X);            // A::f(X) is a friend
    class Y {
      friend void g();           // A::g is a friend
      friend void h(int);        // A::h is a friend
                                 // ::h not considered
      friend void f2<>(int);     // ::f2<>(int) is a friend
    };
  };

  // A::f, A::g and A::h are not visible here
  X x;
  void g() { f(x); }             // definition of A::g
  void f(X) { /* ...  */}        // definition of A::f
  void h(int) { /* ...  */ }     // definition of A::h
  // A::f, A::g and A::h are visible here and known to be friends
}

using A::x;
```

97) this implies that the name of the class or function is unqualified.

```
void h() {
  A::f(x);
  A::X::f(x);                    // error: f is not a member of A::X
  A::X::Y::g();                  // error: g is not a member of A::X::Y
}
```

— *end example*]

7.1306.1 Namespace alias [namespace.alias]

1 A *namespace-alias-definition* declares an alternate name for a namespace according to the following grammar:

namespace-alias:
identifier

namespace-alias-definition:
`namespace` *identifier* `=` *qualified-namespace-specifier* `;`

qualified-namespace-specifier:
nested-name-specifier$_{opt}$ *namespace-name*

2

The *identifier* in a *namespace-alias-definition* is a synonym for the name of the namespace denoted by the *qualified-namespace-specifier* and becomes a *namespace-alias*. [*Note:* When looking up a *namespace-name* in a *namespace-alias-definition*, only namespace names are considered, see 3.1457.1. — *end note*]

1 In a declarative region, a *namespace-alias-definition* can be used to redefine a *namespace-alias* declared in that declarative region to refer only to the namespace to which it already refers. [*Example:* the following declarations are well-formed:

```
namespace Company_with_very_long_name { /* ... */ }
namespace CWVLN = Company_with_very_long_name;
namespace CWVLN = Company_with_very_long_name;          // OK: duplicate
namespace CWVLN = CWVLN;
```

— *end example*]

2

A *namespace-name* or *namespace-alias* shall not be declared as the name of any other entity in the same declarative region. A *namespace-name* defined at global scope shall not be declared as the name of any other entity in any global scope of the program. No diagnostic is required for a violation of this rule by declarations in different translation units.

7.1311.1 The `using` declaration [namespace.udecl]

1

A *using-declaration* introduces a name into the declarative region in which the *using-declaration* appears.

using-declaration:
`using` `typename`$_{opt}$ *nested-name-specifier unqualified-id* `;`
`using` `::` *unqualified-id* `;`

The member name specified in a *using-declaration* is declared in the declarative region in which the *using-declaration* appears. [*Note:* Only the specified name is so declared; specifying an enumeration name in a *using-declaration* does not declare its enumerators in the *using-declaration*'s declarative region. — *end note*] If a *using-declaration* names a constructor (3.1282.0.1), it implicitly declares a set of constructors in the class in which the *using-declaration* appears (12.2400); otherwise the name specified in a *using-declaration* is a synonym for a set of declarations in another namespace or class.

1

Every *using-declaration* is a *declaration* and a *member-declaration* and so can be used in a class definition. [*Example:*

```
struct B {
  void f(char);
  void g(char);
  enum E { e };
  union { int x; };
};

struct D : B {
  using B::f;
  void f(int) { f('c'); }       // calls B::f(char)
  void g(int) { g('c'); }       // recursively calls D::g(int)
};
```

— *end example*]

1

In a *using-declaration* used as a *member-declaration*, the *nested-name-specifier* shall name a base class of the class being defined. If such a *using-declaration* names a constructor, the *nested-name-specifier* shall name a direct base class of the class being defined; otherwise it introduces the set of declarations found by member name lookup (10.52, 3.1282.0.1). [*Example:*

```
class C {
  int g();
};

class D2 : public B {
  using B::f;                   // OK: B is a base of D2
  using B::e;                   // OK: e is an enumerator of base B
  using B::x;                   // OK: x is a union member of base B
  using C::g;                   // error: C isn't a base of D2
};
```

— *end example*]

1

[*Note:* Since destructors do not have names, a *using-declaration* cannot refer to a destructor for a base class. Since specializations of member templates for conversion functions are not found by name lookup, they are not considered when a *using-declaration* specifies a conversion function (14.641.1). — *end note*] If an assignment operator brought from a base class into a derived class scope has the signature of a copy/move assignment operator for the derived class (12.1577), the *using-declaration* does not by itself suppress the implicit declaration of the derived class assignment operator; the copy/move assignment operator from the base class is hidden or overridden by the implicitly-declared copy/move assignment operator of the derived class, as described below.

1 A *using-declaration* shall not name a *template-id*. [*Example:*

```
struct A {
  template <class T> void f(T);
  template <class T> struct X { };
};
struct B : A {
  using A::f<double>;           // ill-formed
  using A::X<int>;              // ill-formed
};
```

— *end example*]

2 A *using-declaration* shall not name a namespace.

3

A *using-declaration* shall not name a scoped enumerator.

1

A *using-declaration* for a class member shall be a *member-declaration*. [*Example:*

```
struct X {
  int i;
  static int s;
};

void f() {
  using X::i;        // error: X::i is a class member
                     // and this is not a member declaration.
  using X::s;        // error: X::s is a class member
                     // and this is not a member declaration.
}
```

— *end example*]

1 Members declared by a *using-declaration* can be referred to by explicit qualification just like other member names (3.1363.0.1). In a *using-declaration*, a prefix `::` refers to the global namespace. [*Example:*

```
void f();

namespace A {
  void g();
}

namespace X {
  using ::f;         // global f
  using A::g;        // A's g
}

void h()
{
  X::f();            // calls ::f
  X::g();            // calls A::g
}
```

— *end example*]

2

A *using-declaration* is a *declaration* and can therefore be used repeatedly where (and only where) multiple declarations are allowed. [*Example:*

```
namespace A {
  int i;
}

namespace A1 {
  using A::i;
  using A::i;        // OK: double declaration
}
```

```
void f() {
  using A::i;
  using A::i;        // error: double declaration
}

struct B {
  int i;
};

struct X : B {
  using B::i;
  using B::i;        // error: double member declaration
};
```

— *end example*]

1 Members added to the namespace after the *using-declaration* are not considered when a use of the name is made. [*Note:* Thus, additional overloads added after the *using-declaration* are ignored, but default function arguments (8.545.1), default template arguments (14.97), and template specializations (14.812.1, 14.1921.1) are considered. — *end note*] [*Example:*

```
namespace A {
  void f(int);
}

using A::f;          // f is a synonym for A::f;
                     // that is, for A::f(int).
namespace A {
  void f(char);
}

void foo() {
  f('a');            // calls f(int),
}                    // even though f(char) exists.

void bar() {
  using A::f;        // f is a synonym for A::f;
                     // that is, for A::f(int) and A::f(char).
  f('a');            // calls f(char)
}
```

— *end example*]

2 [*Note:* Partial specializations of class templates are found by looking up the primary class template and then considering all partial specializations of that template. If a *using-declaration* names a class template, partial specializations introduced after the *using-declaration* are effectively visible because the primary template is visible (14.812.1). — *end note*]

3

Since a *using-declaration* is a declaration, the restrictions on declarations of the same name in the same declarative region (3.464) also apply to *using-declarations*. [*Example:*

```
namespace A {
  int x;
}
```

```
namespace B {
  int i;
  struct g { };
  struct x { };
  void f(int);
  void f(double);
  void g(char);      // OK: hides struct g
}

void func() {
  int i;
  using B::i;        // error: i declared twice
  void f(char);
  using B::f;        // OK: each f is a function
  f(3.5);            // calls B::f(double)
  using B::g;
  g('a');            // calls B::g(char)
  struct g g1;       // g1 has class type B::g
  using B::x;
  using A::x;        // OK: hides struct B::x
  x = 99;            // assigns to A::x
  struct x x1;       // x1 has class type B::x
}
```

— *end example*]

1

If a function declaration in namespace scope or block scope has the same name and the same parameter-type-list (8.360.1) as a function introduced by a *using-declaration*, and the declarations do not declare the same function, the program is ill-formed. If a function template declaration in namespace scope has the same name, parameter-type-list, return type, and template parameter list as a function template introduced by a *using-declaration*, the program is ill-formed. [*Note:* Two *using-declaration*s may introduce functions with the same name and the same parameter-type-list. If, for a call to an unqualified function name, function overload resolution selects the functions introduced by such *using-declaration*s, the function call is ill-formed. [*Example:*

```
namespace B {
  void f(int);
  void f(double);
}
namespace C {
  void f(int);
  void f(double);
  void f(char);
}

void h() {
  using B::f;        // B::f(int) and B::f(double)
  using C::f;        // C::f(int), C::f(double), and C::f(char)
  f('h');            // calls C::f(char)
  f(1);              // error: ambiguous: B::f(int) or C::f(int)?
  void f(int);       // error: f(int) conflicts with C::f(int) and B::f(int)
}
```

— *end example*] — *end note*]

1

When a *using-declaration* brings names from a base class into a derived class scope, member functions and member function templates in the derived class override and/or hide member functions and member function templates with the same name, parameter-type-list (8.360.1), cv-qualification, and *ref-qualifier* (if any) in a base class (rather than conflicting). [*Note:* For *using-declarations* that name a constructor, see 12.2400. — *end note*] [*Example:*

```
struct B {
  virtual void f(int);
  virtual void f(char);
  void g(int);
  void h(int);
};

struct D : B {
  using B::f;
  void f(int);       // OK: D::f(int) overrides B::f(int);

  using B::g;
  void g(char);      // OK

  using B::h;
  void h(int);       // OK: D::h(int) hides B::h(int)
};

void k(D* p)
{
  p->f(1);           // calls D::f(int)
  p->f('a');         // calls B::f(char)
  p->g(1);           // calls B::g(int)
  p->g('a');         // calls D::g(char)
}
```

— *end example*]

1

For the purpose of overload resolution, the functions which are introduced by a *using-declaration* into a derived class will be treated as though they were members of the derived class. In particular, the implicit `this` parameter shall be treated as if it were a pointer to the derived class rather than to the base class. This has no effect on the type of the function, and in all other respects the function remains a member of the base class.

1

The access rules for inheriting constructors are specified in 12.2400; otherwise all instances of the name mentioned in a *using-declaration* shall be accessible. In particular, if a derived class uses a *using-declaration* to access a member of a base class, the member name shall be accessible. If the name is that of an overloaded member function, then all functions named shall be accessible. The base class members mentioned by a *using-declaration* shall be visible in the scope of at least one of the direct base classes of the class where the *using-declaration* is specified. [*Note:* Because a *using-declaration* designates a base class member (and not a member subobject or a member function of a base class subobject), a *using-declaration* cannot be used to resolve inherited member ambiguities. For example,

```
struct A { int x(); };
struct B : A { };
struct C : A {
```

```
  using A::x;
  int x(int);
};

struct D : B, C {
  using C::x;
  int x(double);
};
int f(D* d) {
  return d->x();     // ambiguous: B::x or C::x
}
```

— *end note*]

1

The alias created by the *using-declaration* has the usual accessibility for a *member-declaration*. [*Note:* A *using-declaration* that names a constructor does not create aliases; see 12.2400 for the pertinent accessibility rules. — *end note*] [*Example:*

```
class A {
private:
    void f(char);
public:
    void f(int);
protected:
    void g();
};

class B : public A {
  using A::f;       // error: A::f(char) is inaccessible
public:
  using A::g;       // B::g is a public synonym for A::g
};
```

— *end example*]

1

If a *using-declaration* uses the keyword `typename` and specifies a dependent name (14.1257.1), the name introduced by the *using-declaration* is treated as a *typedef-name* (7.235.1).

7.1416.1 Using directive [namespace.udir]

using-directive:
attribute-specifier-seq$_{opt}$ `using namespace` *nested-name-specifier*$_{opt}$ *namespace-name* ;

1

A *using-directive* shall not appear in class scope, but may appear in namespace scope or in block scope. [*Note:* When looking up a *namespace-name* in a *using-directive*, only namespace names are considered, see 3.1457.1. — *end note*] The optional *attribute-specifier-seq* appertains to the *using-directive*.

1

A *using-directive* specifies that the names in the nominated namespace can be used in the scope in which the *using-directive* appears after the *using-directive*. During unqualified name lookup (3.952.1), the names appear as if they were declared in the nearest enclosing namespace which contains both the *using-directive* and the nominated namespace. [*Note:* In this context, "contains" means "contains directly or indirectly". — *end note*]

1

A *using-directive* does not add any members to the declarative region in which it appears. [*Example:*

```
namespace A {
  int i;
  namespace B {
    namespace C {
      int i;
    }
    using namespace A::B::C;
    void f1() {
      i = 5;        // OK, C::i visible in B and hides A::i
    }
  }
  namespace D {
    using namespace B;
    using namespace C;
    void f2() {
      i = 5;        // ambiguous, B::C::i or A::i?
    }
  }
  void f3() {
    i = 5;          // uses A::i
  }
}
void f4() {
  i = 5;            // ill-formed; neither i is visible
}
```

— *end example*]

1

For unqualified lookup (3.952.1), the *using-directive* is transitive: if a scope contains a *using-directive* that nominates a second namespace that itself contains *using-directive*s, the effect is as if the *using-directive*s from the second namespace also appeared in the first. [*Note:* For qualified lookup, see 3.1363.0.1. — *end note*] [*Example:*

```
namespace M {
  int i;
}

namespace N {
  int i;
  using namespace M;
}

void f() {
  using namespace N;
  i = 7;            // error: both M::i and N::i are visible
}
```

For another example,

```
namespace A {
  int i;
}
```

```
namespace B {
  int i;
  int j;
  namespace C {
    namespace D {
      using namespace A;
      int j;
      int k;
      int a = i;    // B::i hides A::i
    }
    using namespace D;
    int k = 89;     // no problem yet
    int l = k;      // ambiguous: C::k or D::k
    int m = i;      // B::i hides A::i
    int n = j;      // D::j hides B::j
  }
}
```

— *end example*]

1 If a namespace is extended by an *extension-namespace-definition* after a *using-directive* for that namespace is given, the additional members of the extended namespace and the members of namespaces nominated by *using-directive*s in the *extension-namespace-definition* can be used after the *extension-namespace-definition*.

2 If name lookup finds a declaration for a name in two different namespaces, and the declarations do not declare the same entity and do not declare functions, the use of the name is ill-formed. [*Note:* In particular, the name of a variable, function or enumerator does not hide the name of a class or enumeration declared in a different namespace. For example,

```
namespace A {
  class X { };
  extern "C"   int g();
  extern "C++" int h();
}
namespace B {
  void X(int);
  extern "C"   int g();
  extern "C++" int h(int);
}
using namespace A;
using namespace B;

void f() {
  X(1);             // error: name X found in two namespaces
  g();              // okay: name g refers to the same entity
  h();              // okay: overload resolution selects A::h
}
```

— *end note*]

1 During overload resolution, all functions from the transitive search are considered for argument matching. The set of declarations found by the transitive search is unordered. [*Note:* In particular, the order in which namespaces were considered and the relationships among the namespaces implied by the *using-directive*s do not cause preference to be given to any of the declarations found by the search. — *end note*] An ambiguity exists if the best match finds two functions with the same signature, even if one is in a namespace reachable

through *using-directives* in the namespace of the other.[98] [*Example:*

```
namespace D {
  int d1;
  void f(char);
}
using namespace D;

int d1;             // OK: no conflict with D::d1

namespace E {
  int e;
  void f(int);
}

namespace D {       // namespace extension
  int d2;
  using namespace E;
  void f(int);
}

void f() {
  d1++;             // error: ambiguous ::d1 or D::d1?
  ::d1++;           // OK
  D::d1++;          // OK
  d2++;             // OK: D::d2
  e++;              // OK: E::e
  f(1);             // error: ambiguous: D::f(int) or E::f(int)?
  f('a');           // OK: D::f(char)
}
```

— *end example*]

7.1473 The `asm` declaration [dcl.asm]

1

An `asm` declaration has the form

asm-definition:
`asm` (*string-literal*) ;

The `asm` declaration is conditionally-supported; its meaning is implementation-defined. [*Note:* Typically it is used to pass information through the implementation to an assembler. — *end note*]

7.1477 Linkage specifications [dcl.link]

1

All function types, function names with external linkage, and variable names with external linkage have a *language linkage*. [*Note:* Some of the properties associated with an entity with language linkage are specific to each implementation and are not described here. For example, a particular language linkage may be associated with a particular form of representing names of objects and functions with external linkage, or with a particular calling convention, etc. — *end note*] The default language linkage of all function types, function names, and variable names is C++ language linkage. Two function types with different language linkages are distinct types even if they are otherwise identical.

98) During name lookup in a class hierarchy, some ambiguities may be resolved by considering whether one member hides the other along some paths (10.52). There is no such disambiguation when considering the set of names found as a result of following *using-directives*.

1

Linkage (3.1459) between C++ and non-C++ code fragments can be achieved using a *linkage-specification*:

linkage-specification:
`extern` *string-literal* `{` *declaration-seq*$_{opt}$`}`
`extern` *string-literal declaration*

The *string-literal* indicates the required language linkage. This International Standard specifies the semantics for the *string-literals* `"C"` and `"C++"`. Use of a *string-literal* other than `"C"` or `"C++"` is conditionally-supported, with implementation-defined semantics. [*Note:* Therefore, a linkage-specification with a *string-literal* that is unknown to the implementation requires a diagnostic. — *end note*] [*Note:* It is recommended that the spelling of the *string-literal* be taken from the document defining that language. For example, `Ada` (not `ADA`) and `Fortran` or `FORTRAN`, depending on the vintage. — *end note*]

1

Every implementation shall provide for linkage to functions written in the C programming language, `"C"`, and linkage to C++ functions, `"C++"`. [*Example:*

```
complex sqrt(complex);          // C++ linkage by default
extern "C" {
  double sqrt(double);          // C linkage
}
```

— *end example*]

1

Linkage specifications nest. When linkage specifications nest, the innermost one determines the language linkage. A linkage specification does not establish a scope. A *linkage-specification* shall occur only in namespace scope (3.464). In a *linkage-specification*, the specified language linkage applies to the function types of all function declarators, function names with external linkage, and variable names with external linkage declared within the *linkage-specification*. [*Example:*

```
extern "C" void f1(void(*pf)(int));
                                // the name f1 and its function type have C language
                                // linkage; pf is a pointer to a C function
extern "C" typedef void FUNC();
FUNC f2;                        // the name f2 has C++ language linkage and the
                                // function's type has C language linkage
extern "C" FUNC f3;             // the name of function f3 and the function's type
                                // have C language linkage
void (*pf2)(FUNC*);             // the name of the variable pf2 has C++ linkage and
                                // the type of pf2 is pointer to C++ function that
                                // takes one parameter of type pointer to C function
extern "C" {
  static void f4();             // the name of the function f4 has
                                // internal linkage (not C language
                                // linkage) and the function's type
                                // has C language linkage.
}

extern "C" void f5() {
  extern void f4();             // OK: Name linkage (internal)
                                // and function type linkage (C
                                // language linkage) obtained from
                                // previous declaration.
}
```

```
extern void f4();                // OK: Name linkage (internal)
                                 // and function type linkage (C
                                 // language linkage) obtained from
                                 // previous declaration.

void f6() {
  extern void f4();              // OK: Name linkage (internal)
                                 // and function type linkage (C
                                 // language linkage) obtained from
                                 // previous declaration.
}
```

— *end example*] A C language linkage is ignored in determining the language linkage of the names of class members and the function type of class member functions. [*Example:*

```
extern "C" typedef void FUNC_c();
class C {
   void mf1(FUNC_c*);            // the name of the function mf1 and the member
                                 // function's type have C++ language linkage; the
                                 // parameter has type pointer to C function
   FUNC_c mf2;                   // the name of the function mf2 and the member
                                 // function's type have C++ language linkage
   static FUNC_c* q;             // the name of the data member q has C++ language
                                 // linkage and the data member's type is pointer to
                                 // C function
};

extern "C" {
  class X {
  void mf();                     // the name of the function mf and the member
                                 // function's type have C++ language linkage
  void mf2(void(*)());           // the name of the function mf2 has C++ language
                                 // linkage; the parameter has type pointer to
                                 // C function
  };
}
```

— *end example*]

1

If two declarations declare functions with the same name and *parameter-type-list* (8.360.1) to be members of the same namespace or declare objects with the same name to be members of the same namespace and the declarations give the names different language linkages, the program is ill-formed; no diagnostic is required if the declarations appear in different translation units. Except for functions with C++ linkage, a function declaration without a linkage specification shall not precede the first linkage specification for that function. A function can be declared without a linkage specification after an explicit linkage specification has been seen; the linkage explicitly specified in the earlier declaration is not affected by such a function declaration.

1

At most one function with a particular name can have C language linkage. Two declarations for a function with C language linkage with the same function name (ignoring the namespace names that qualify it) that appear in different namespace scopes refer to the same function. Two declarations for a variable with C language linkage with the same name (ignoring the namespace names that qualify it) that appear in different namespace scopes refer to the same variable. An entity with C language linkage shall not be declared with the same name as an entity in global scope, unless both declarations denote the same entity; no diagnostic

is required if the declarations appear in different translation units. A variable with C language linkage shall not be declared with the same name as a function with C language linkage (ignoring the namespace names that qualify the respective names); no diagnostic is required if the declarations appear in different translation units. [*Note:* Only one definition for an entity with a given name with C language linkage may appear in the program (see 3.91); this implies that such an entity must not be defined in more than one namespace scope. — *end note*] [*Example:*

```
int x;
namespace A {
  extern "C" int f();
  extern "C" int g() { return 1; }
  extern "C" int h();
  extern "C" int x();                // ill-formed: same name as global-space object x
}

namespace B {
  extern "C" int f();                // A::f and B::f refer to the same function
  extern "C" int g() { return 1; }   // ill-formed, the function g
                                     // with C language linkage has two definitions
}

int A::f() { return 98; }            //definition for the function f with C language linkage
extern "C" int h() { return 97; }    // definition for the function h with C language linkage
                                     // A::h and ::h refer to the same function
```

— *end example*]

1

A declaration directly contained in a *linkage-specification* is treated as if it contains the `extern` specifier (7.49.1) for the purpose of determining the linkage of the declared name and whether it is a definition. Such a declaration shall not specify a storage class. [*Example:*

```
extern "C" double f();
static double f();                   // error
extern "C" int i;                    // declaration
extern "C" {
  int i;                             // definition
}
extern "C" static void g();          // error
```

— *end example*]

1 [*Note:* Because the language linkage is part of a function type, when indirecting through a pointer to C function, the function to which the resulting lvalue refers is considered a C function. — *end note*]

2

Linkage from C++ to objects defined in other languages and to objects defined in C++ from other languages is implementation-defined and language-dependent. Only where the object layout strategies of two language implementations are similar enough can such linkage be achieved.

7.1546 Attributes [dcl.attr]

7.1546.1 Attribute syntax and semantics [dcl.attr.grammar]

1 Attributes specify additional information for various source constructs such as types, variables, names, blocks, or translation units.

attribute-specifier-seq:
  attribute-specifier-seq$_{opt}$ *attribute-specifier*

attribute-specifier:
  `[ [` *attribute-list* `] ]`
  alignment-specifier

alignment-specifier:
  `alignas (` *type-id* `...`$_{opt}$`)`
  `alignas (` *constant-expression* `...`$_{opt}$`)`

attribute-list:
  attribute$_{opt}$
  attribute-list `,` *attribute*$_{opt}$
  attribute `...`
  attribute-list `,` *attribute* `...`

attribute:
  attribute-token attribute-argument-clause$_{opt}$

attribute-token:
  identifier
  attribute-scoped-token

attribute-scoped-token:
  attribute-namespace `::` *identifier*

attribute-namespace:
  identifier

attribute-argument-clause:
  `(` *balanced-token-seq* `)`

balanced-token-seq:
  balanced-token$_{opt}$
  balanced-token-seq balanced-token

balanced-token:
  `(` *balanced-token-seq* `)`
  `[` *balanced-token-seq* `]`
  `{` *balanced-token-seq* `}`
  any *token* other than a parenthesis, a bracket, or a brace

2 [*Note:* For each individual attribute, the form of the *balanced-token-seq* will be specified. — *end note*]

3

In an *attribute-list*, an ellipsis may appear only if that *attribute*'s specification permits it. An *attribute* followed by an ellipsis is a pack expansion (14.703.1). An *attribute-specifier* that contains no *attribute*s has no effect. The order in which the *attribute-tokens* appear in an *attribute-list* is not significant. If a keyword (2.287) or an alternative token (2.236) that satisfies the syntactic requirements of an *identifier* (2.268) is contained in an *attribute-token*, it is considered an identifier. No name lookup (3.923) is performed on any of the identifiers contained in an *attribute-token*. The *attribute-token* determines additional requirements on the *attribute-argument-clause* (if any). The use of an *attribute-scoped-token* is conditionally-supported, with implementation-defined behavior. [*Note:* Each implementation should choose a distinctive name for the *attribute-namespace* in an *attribute-scoped-token*. — *end note*]

1

Each *attribute-specifier-seq* is said to *appertain* to some entity or statement, identified by the syntactic context where it appears (Clause 6, Clause 7, Clause 8). If an *attribute-specifier-seq* that appertains to some entity or statement contains an *attribute* that is not allowed to apply to that entity or statement, the program is ill-formed. If an *attribute-specifier-seq* appertains to a friend declaration (11.94), that declaration shall be a definition. No *attribute-specifier-seq* shall appertain to an explicit instantiation (14.1841.1).

1

For an *attribute-token* not specified in this International Standard, the behavior is implementation-defined.

1

Two consecutive left square bracket tokens shall appear only when introducing an *attribute-specifier*. [*Note:* If two consecutive left square brackets appear where an *attribute-specifier* is not allowed, the program is ill-formed even if the brackets match an alternative grammar production. — *end note*] [*Example:*

```
int p[10];
void f() {
  int x = 42, y[5];
  int(p[[x] { return x; }()]);   // error: invalid attribute on a nested
                                 // declarator-id and not a function-style cast of
                                 // an element of p.
  y[[] { return 2; }()] = 2;     // error even though attributes are not allowed
                                 // in this context.
}
```

— *end example*]

7.1556.1 Alignment specifier [dcl.align]

1

An *alignment-specifier* may be applied to a variable or to a class data member, but it shall not be applied to a bit-field, a function parameter, an *exception-declaration* (15.146), or a variable declared with the `register` storage class specifier. An *alignment-specifier* may also be applied to the declaration or definition of a class (in an *elaborated-type-specifier* (7.808.0.1) or *class-head* (Clause 9), respectively) and to the declaration or definition of an enumeration (in an *opaque-enum-declaration* or *enum-head*, respectively (7.1001)). An *alignment-specifier* with an ellipsis is a pack expansion (14.703.1).

1

When the *alignment-specifier* is of the form `alignas(` *constant-expression* `)`:

— the *constant-expression* shall be an integral constant expression

— if the constant expression evaluates to a fundamental alignment, the alignment requirement of the declared entity shall be the specified fundamental alignment

— if the constant expression evaluates to an extended alignment and the implementation supports that alignment in the context of the declaration, the alignment of the declared entity shall be that alignment

— if the constant expression evaluates to an extended alignment and the implementation does not support that alignment in the context of the declaration, the program is ill-formed

— if the constant expression evaluates to zero, the alignment specifier shall have no effect

— otherwise, the program is ill-formed.

1 When the *alignment-specifier* is of the form `alignas(` *type-id* `)`, it shall have the same effect as `alignas(` `alignof(`*type-id* `))` (5.2747.1).

2 When multiple *alignment-specifier*s are specified for an entity, the alignment requirement shall be set to the strictest specified alignment.

3

The combined effect of all *alignment-specifier*s in a declaration shall not specify an alignment that is less strict than the alignment that would be required for the entity being declared if all *alignment-specifier*s were omitted (including those in other declarations).

1 If the defining declaration of an entity has an *alignment-specifier*, any non-defining declaration of that entity shall either specify equivalent alignment or have no *alignment-specifier*. Conversely, if any declaration of an entity has an *alignment-specifier*, every defining declaration of that entity shall specify an equivalent alignment. No diagnostic is required if declarations of an entity have different *alignment-specifier*s in different translation units.

[*Example:*

```
// Translation unit #1:
struct S { int x; } s, p = &s;

// Translation unit #2:
struct alignas(16) S;          // error: definition of S lacks alignment; no
extern S* p;                   // diagnostic required
```

— *end example*]

2 [*Example:* An aligned buffer with an alignment requirement of `A` and holding `N` elements of type `T` other than `char`, `signed char`, or `unsigned char` can be declared as:

```
alignas(T) alignas(A) T buffer[N];
```

Specifying `alignas(T)` ensures that the final requested alignment will not be weaker than `alignof(T)`, and therefore the program will not be ill-formed. — *end example*]

3 [*Example:*

```
alignas(double) void f();                        // error: alignment applied to function
alignas(double) unsigned char c[sizeof(double)]; // array of characters, suitably aligned for a double
extern unsigned char c[sizeof(double)];          // no alignas necessary
alignas(float)
  extern unsigned char c[sizeof(double)];        // error: different alignment in declaration
```

— *end example*]

7.1569.1 Noreturn attribute [dcl.attr.noreturn]

1

The *attribute-token* `noreturn` specifies that a function does not return. It shall appear at most once in each *attribute-list* and no *attribute-argument-clause* shall be present. The attribute may be applied to the *declarator-id* in a function declaration. The first declaration of a function shall specify the `noreturn` attribute if any declaration of that function specifies the `noreturn` attribute. If a function is declared with the `noreturn` attribute in one translation unit and the same function is declared without the `noreturn` attribute in another translation unit, the program is ill-formed; no diagnostic required.

1

If a function `f` is called where `f` was previously declared with the `noreturn` attribute and `f` eventually returns, the behavior is undefined. [*Note:* The function may terminate by throwing an exception. — *end note*] [*Note:* Implementations are encouraged to issue a warning if a function marked `[[noreturn]]` might return. — *end note*]

1 [*Example:*

```
[[ noreturn ]] void f() {
  throw "error";        // OK
```

```
}

[[ noreturn ]] void q(int i) { // behavior is undefined if called with an argument <= 0
  if (i > 0)
    throw "positive";
}
```

— *end example*]

7.1574.1 Carries dependency attribute [dcl.attr.depend]

1

The *attribute-token* `carries_dependency` specifies dependency propagation into and out of functions. It shall appear at most once in each *attribute-list* and no *attribute-argument-clause* shall be present. The attribute may be applied to the *declarator-id* of a *parameter-declaration* in a function declaration or lambda, in which case it specifies that the initialization of the parameter carries a dependency to (1.859) each lvalue-to-rvalue conversion (4.127) of that object. The attribute may also be applied to the *declarator-id* of a function declaration, in which case it specifies that the return value, if any, carries a dependency to the evaluation of the function call expression.

1 The first declaration of a function shall specify the `carries_dependency` attribute for its *declarator-id* if any declaration of the function specifies the `carries_dependency` attribute. Furthermore, the first declaration of a function shall specify the `carries_dependency` attribute for a parameter if any declaration of that function specifies the `carries_dependency` attribute for that parameter. If a function or one of its parameters is declared with the `carries_dependency` attribute in its first declaration in one translation unit and the same function or one of its parameters is declared without the `carries_dependency` attribute in its first declaration in another translation unit, the program is ill-formed; no diagnostic required.

2 [*Note:* The `carries_dependency` attribute does not change the meaning of the program, but may result in generation of more efficient code. — *end note*]

3 [*Example:*

```
/* Translation unit A. */

struct foo { int* a; int* b; };
std::atomic<struct foo *> foo_head[10];
int foo_array[10][10];

[[carries_dependency]] struct foo* f(int i) {
  return foo_head[i].load(memory_order_consume);
}

int g(int* x, int* y [[carries_dependency]]) {
  return kill_dependency(foo_array[*x][*y]);
}

/* Translation unit B. */

[[carries_dependency]] struct foo* f(int i);
int g(int* x, int* y [[carries_dependency]]);

int c = 3;

void h(int i) {
  struct foo* p;
```

```
  p = f(i);
  do_something_with(g(&c, p->a));
  do_something_with(g(p->a, &c));
}
```

4 The `carries_dependency` attribute on function `f` means that the return value carries a dependency out of `f`, so that the implementation need not constrain ordering upon return from `f`. Implementations of `f` and its caller may choose to preserve dependencies instead of emitting hardware memory ordering instructions (a.k.a. fences).

5 Function `g`'s second parameter has a `carries_dependency` attribute, but its first parameter does not. Therefore, function `h`'s first call to `g` carries a dependency into `g`, but its second call does not. The implementation might need to insert a fence prior to the second call to `g`.

— *end example*]

7.1576.1 Deprecated attribute [dcl.attr.deprecated]

1

The *attribute-token* `deprecated` can be used to mark names and entities whose use is still allowed, but is discouraged for some reason. [*Note:* in particular, `deprecated` is appropriate for names and entities that are deemed obsolescent or unsafe. — *end note*] It shall appear at most once in each *attribute-list*. An *attribute-argument-clause* may be present and, if present, it shall have the form:

```
( string-literal )
```

[*Note:* the *string-literal* in the *attribute-argument-clause* could be used to explain the rationale for deprecation and/or to suggest a replacing entity. — *end note*]

1

The attribute may be applied to the declaration of a class, a *typedef-name*, a variable, a non-static data member, a function, an enumeration, or a template specialization.

1

A name or entity declared without the `deprecated` attribute can later be re-declared with the attribute and vice-versa. [*Note:* Thus, an entity initially declared without the attribute can be marked as deprecated by a subsequent redeclaration. However, after an entity is marked as deprecated, later redeclarations do not un-deprecate the entity. — *end note*] Redeclarations using different forms of the attribute (with or without the *attribute-argument-clause* or with different *attribute-argument-clauses*) are allowed.

1

[*Note:* Implementations may use the `deprecated` attribute to produce a diagnostic message in case the program refers to a name or entity other than to declare it, after a declaration that specifies the attribute. The diagnostic message may include the text provided within the *attribute-argument-clause* of any `deprecated` attribute applied to the name or entity. — *end note*]

8 Declarators [dcl.decl]

1

A declarator declares a single variable, function, or type, within a declaration. The *init-declarator-list* appearing in a declaration is a comma-separated sequence of declarators, each of which can have an initializer.

init-declarator-list:
  init-declarator
  init-declarator-list **,** *init-declarator*

init-declarator:
  declarator initializer$_{opt}$

1

The three components of a *simple-declaration* are the attributes (7.1546), the specifiers (*decl-specifier-seq*; 7.40) and the declarators (*init-declarator-list*). The specifiers indicate the type, storage class or other properties of the entities being declared. The declarators specify the names of these entities and (optionally) modify the type of the specifiers with operators such as `*` (pointer to) and `()` (function returning). Initial values can also be specified in a declarator; initializers are discussed in 8.821 and 12.1011.

1

Each *init-declarator* in a declaration is analyzed separately as if it was in a declaration by itself.[99]

1

Declarators have the syntax

declarator:
  ptr-declarator
  noptr-declarator parameters-and-qualifiers trailing-return-type

ptr-declarator:
  noptr-declarator
  ptr-operator ptr-declarator

99) A declaration with several declarators is usually equivalent to the corresponding sequence of declarations each with a single declarator. That is

```
T D1, D2, ...  Dn;
```

is usually equivalent to

```
T D1; T D2; ...  T Dn;
```

where `T` is a *decl-specifier-seq* and each `Di` is an *init-declarator*. An exception occurs when a name introduced by one of the *declarator*s hides a type name used by the *decl-specifiers*, so that when the same *decl-specifiers* are used in a subsequent declaration, they do not have the same meaning, as in

```
struct S  ...  ;
S S, T; // declare two instances of struct S
```

which is not equivalent to

```
struct S  ...  ;
S S;
S T; // error
```

Another exception occurs when `T` is `auto` (7.832.0.1), for example:

```
auto i = 1, j = 2.0; // error: deduced types for i and j do not match
```

as opposed to

```
auto i = 1; // OK: i deduced to have type int
auto j = 2.0; // OK: j deduced to have type double
```

noptr-declarator:
  declarator-id attribute-specifier-seq$_{opt}$
  noptr-declarator parameters-and-qualifiers
  noptr-declarator `[` *constant-expression*$_{opt}$`]` *attribute-specifier-seq*$_{opt}$
  `(` *ptr-declarator* `)`

parameters-and-qualifiers:
  `(` *parameter-declaration-clause* `)` *cv-qualifier-seq*$_{opt}$
    ref-qualifier$_{opt}$ *exception-specification*$_{opt}$ *attribute-specifier-seq*$_{opt}$

trailing-return-type:
  `->` *trailing-type-specifier-seq abstract-declarator*$_{opt}$

ptr-operator:
  `*` *attribute-specifier-seq*$_{opt}$ *cv-qualifier-seq*$_{opt}$
  `&` *attribute-specifier-seq*$_{opt}$
  `&&` *attribute-specifier-seq*$_{opt}$
  nested-name-specifier `*` *attribute-specifier-seq*$_{opt}$ *cv-qualifier-seq*$_{opt}$

cv-qualifier-seq:
  cv-qualifier cv-qualifier-seq$_{opt}$

cv-qualifier:
  `const`
  `volatile`

ref-qualifier:
  `&`
  `&&`

declarator-id:
  `...`$_{opt}$ *id-expression*

1

The optional *attribute-specifier-seq* in a *trailing-return-type* appertains to the indicated return type. The *type-id* in a *trailing-return-type* includes the longest possible sequence of *abstract-declarator*s. [*Note:* This resolves the ambiguous binding of array and function declarators. [*Example:*

```
auto f()->int(*)[4];   // function returning a pointer to array[4] of int
                       // not function returning array[4] of pointer to int
```

— *end example*] — *end note*]

8.32 Type names [dcl.name]

1

To specify type conversions explicitly, and as an argument of `sizeof`, `alignof`, `new`, or `typeid`, the name of a type shall be specified. This can be done with a *type-id*, which is syntactically a declaration for a variable or function of that type that omits the name of the entity.

type-id:
  type-specifier-seq abstract-declarator$_{opt}$

abstract-declarator:
  ptr-abstract-declarator
  noptr-abstract-declarator$_{opt}$ *parameters-and-qualifiers trailing-return-type*
  abstract-pack-declarator

ptr-abstract-declarator:
  noptr-abstract-declarator
  ptr-operator ptr-abstract-declarator$_{opt}$

noptr-abstract-declarator:
noptr-abstract-declarator$_{opt}$ *parameters-and-qualifiers*
noptr-abstract-declarator$_{opt}$ `[` *constant-expression*$_{opt}$ `]` *attribute-specifier-seq*$_{opt}$
`(` *ptr-abstract-declarator* `)`
abstract-pack-declarator:
noptr-abstract-pack-declarator
ptr-operator abstract-pack-declarator
noptr-abstract-pack-declarator:
noptr-abstract-pack-declarator parameters-and-qualifiers
noptr-abstract-pack-declarator `[` *constant-expression*$_{opt}$ `]` *attribute-specifier-seq*$_{opt}$
`...`

It is possible to identify uniquely the location in the *abstract-declarator* where the identifier would appear if the construction were a declarator in a declaration. The named type is then the same as the type of the hypothetical identifier. [*Example:*

```
int                 // int i
int *               // int *pi
int *[3]            // int *p[3]
int (*)[3]          // int (*p3i)[3]
int *()             // int *f()
int (*)(double)     // int (*pf)(double)
```

name respectively the types "`int`," "pointer to `int`," "array of 3 pointers to `int`," "pointer to array of 3 `int`," "function of (no parameters) returning pointer to `int`," and "pointer to a function of (`double`) returning `int`." — *end example*]

1 A type can also be named (often more easily) by using a *typedef* (7.235.1).

8.38 Ambiguity resolution [dcl.ambig.res]

1

The ambiguity arising from the similarity between a function-style cast and a declaration mentioned in 6.556 can also occur in the context of a declaration. In that context, the choice is between a function declaration with a redundant set of parentheses around a parameter name and an object declaration with a function-style cast as the initializer. Just as for the ambiguities mentioned in 6.556, the resolution is to consider any construct that could possibly be a declaration a declaration. [*Note:* A declaration can be explicitly disambiguated by a nonfunction-style cast, by an `=` to indicate initialization or by removing the redundant parentheses around the parameter name. — *end note*] [*Example:*

```
struct S {
  S(int);
};

void foo(double a) {
  S w(int(a));       // function declaration
  S x(int());        // function declaration
  S y((int)a);       // object declaration
  S z = int(a);      // object declaration
}
```

— *end example*]

1

The ambiguity arising from the similarity between a function-style cast and a *type-id* can occur in different contexts. The ambiguity appears as a choice between a function-style cast expression and a declaration of a

type. The resolution is that any construct that could possibly be a *type-id* in its syntactic context shall be considered a *type-id*.

1 [*Example:*

```
#include <cstddef>
char* p;
void* operator new(std::size_t, int);
void foo()  {
  const int x = 63;
  new (int(*p)) int;            // new-placement expression
  new (int(*[x]));              // new type-id
}
```

2 For another example,

```
template <class T>
struct S {
  T* p;
};
S<int()> x;                     // type-id
S<int(1)> y;                    // expression (ill-formed)
```

3 For another example,

```
void foo() {
  sizeof(int(1));               // expression
  sizeof(int());                // type-id (ill-formed)
}
```

4 For another example,

```
void foo() {
  (int(1));                     // expression
  (int())1;                     // type-id (ill-formed)
}
```

— *end example*]

5

Another ambiguity arises in a *parameter-declaration-clause* of a function declaration, or in a *type-id* that is the operand of a `sizeof` or `typeid` operator, when a *type-name* is nested in parentheses. In this case, the choice is between the declaration of a parameter of type pointer to function and the declaration of a parameter with redundant parentheses around the *declarator-id*. The resolution is to consider the *type-name* as a *simple-type-specifier* rather than a *declarator-id*. [*Example:*

```
class C { };
void f(int(C)) { }              // void f(int(*fp)(C c)) { }
                                // not: void f(int C);

int g(C);

void foo() {
  f(1);                         // error: cannot convert 1 to function pointer
  f(g);                         // OK
}
```

For another example,

```
class C { };
void h(int *(C[10]));          // void h(int *(*_fp)(C _parm[10]));
                               // not: void h(int *C[10]);
```

— *end example*]

8.56 Meaning of declarators [dcl.meaning]

1

A list of declarators appears after an optional (Clause 7) *decl-specifier-seq* (7.40). Each declarator contains exactly one *declarator-id*; it names the identifier that is declared. An *unqualified-id* occurring in a *declarator-id* shall be a simple *identifier* except for the declaration of some special functions (12.30, 12.659, 12.807, 13.1122) and for the declaration of template specializations or partial specializations (14.1614). When the *declarator-id* is qualified, the declaration shall refer to a previously declared member of the class or namespace to which the qualifier refers (or, in the case of a namespace, of an element of the inline namespace set of that namespace (7.1149.1)) or to a specialization thereof; the member shall not merely have been introduced by a *using-declaration* in the scope of the class or namespace nominated by the *nested-name-specifier* of the *declarator-id*. The *nested-name-specifier* of a qualified *declarator-id* shall not begin with a *decltype-specifier*. [*Note:* If the qualifier is the global `::` scope resolution operator, the *declarator-id* refers to a name declared in the global namespace scope. — *end note*] The optional *attribute-specifier-seq* following a *declarator-id* appertains to the entity that is declared.

1 A `static`, `thread_local`, `extern`, `register`, `mutable`, `friend`, `inline`, `virtual`, or `typedef` specifier applies directly to each *declarator-id* in an *init-declarator-list*; the type specified for each *declarator-id* depends on both the *decl-specifier-seq* and its *declarator*.

2

Thus, a declaration of a particular identifier has the form

```
T D
```

where `T` is of the form *attribute-specifier-seq*$_{opt}$ *decl-specifier-seq* and `D` is a declarator. Following is a recursive procedure for determining the type specified for the contained *declarator-id* by such a declaration.

1 First, the *decl-specifier-seq* determines a type. In a declaration

```
T D
```

the *decl-specifier-seq* `T` determines the type `T`. [*Example:* in the declaration

```
int unsigned i;
```

the type specifiers `int unsigned` determine the type "`unsigned int`" (7.678.0.1). — *end example*]

2 In a declaration *attribute-specifier-seq*$_{opt}$ `T D` where `D` is an unadorned identifier the type of this identifier is "`T`".

3

In a declaration `T D` where `D` has the form

(D1)

the type of the contained *declarator-id* is the same as that of the contained *declarator-id* in the declaration

```
T D1
```

Parentheses do not alter the type of the embedded *declarator-id*, but they can alter the binding of complex declarators.

8.94.1 Pointers [dcl.ptr]

1

In a declaration `T D` where `D` has the form

`*` *attribute-specifier-seq*$_{opt}$ *cv-qualifier-seq*$_{opt}$`D1`

and the type of the identifier in the declaration `T D1` is "*derived-declarator-type-list* `T`," then the type of the identifier of `D` is "*derived-declarator-type-list cv-qualifier-seq* pointer to `T`." The *cv-qualifier*s apply to the pointer and not to the object pointed to. Similarly, the optional *attribute-specifier-seq* (7.1546.1) appertains to the pointer and not to the object pointed to.

1 [*Example:* the declarations

```
const int ci = 10, *pc = &ci, *const cpc = pc, **ppc;
int i, *p, *const cp = &i;
```

declare `ci`, a constant integer; `pc`, a pointer to a constant integer; `cpc`, a constant pointer to a constant integer; `ppc`, a pointer to a pointer to a constant integer; `i`, an integer; `p`, a pointer to integer; and `cp`, a constant pointer to integer. The value of `ci`, `cpc`, and `cp` cannot be changed after initialization. The value of `pc` can be changed, and so can the object pointed to by `cp`. Examples of some correct operations are

```
i = ci;
*cp = ci;
pc++;
pc = cpc;
pc = p;
ppc = &pc;
```

Examples of ill-formed operations are

```
ci = 1;          // error
ci++;            // error
*pc = 2;         // error
cp = &ci;        // error
cpc++;           // error
p = pc;          // error
ppc = &p;        // error
```

Each is unacceptable because it would either change the value of an object declared `const` or allow it to be changed through a cv-unqualified pointer later, for example:

```
*ppc = &ci;      // OK, but would make p point to ci ...
                 // ... because of previous error
*p = 5;          // clobber ci
```

— *end example*]

2 See also 5.3483 and 8.821.

3

[*Note:* Forming a pointer to reference type is ill-formed; see 8.97.1. Forming a pointer to function type is ill-formed if the function type has *cv-qualifier*s or a *ref-qualifier*; see 8.360.1. Since the address of a bit-field (9.921) cannot be taken, a pointer can never point to a bit-field. — *end note*]

8.97.1 References [dcl.ref]

1

In a declaration `T D` where `D` has either of the forms

& *attribute-specifier-seq*$_{opt}$D1
&& *attribute-specifier-seq*$_{opt}$D1

and the type of the identifier in the declaration `T D1` is "*derived-declarator-type-list* `T`," then the type of the identifier of `D` is "*derived-declarator-type-list* reference to `T`." The optional *attribute-specifier-seq* appertains to the reference type. Cv-qualified references are ill-formed except when the cv-qualifiers are introduced through the use of a *typedef-name* (7.235.1, 14.97) or *decltype-specifier* (7.678.0.1), in which case the cv-qualifiers are ignored. [*Example:*

```
typedef int& A;
const A aref = 3;    // ill-formed; lvalue reference to non-const initialized with rvalue
```

The type of `aref` is "lvalue reference to `int`", not "lvalue reference to `const int`". — *end example*] [*Note:* A reference can be thought of as a name of an object. — *end note*] A declarator that specifies the type "reference to *cv* `void`" is ill-formed.

1

A reference type that is declared using `&` is called an *lvalue reference*, and a reference type that is declared using `&&` is called an *rvalue reference*. Lvalue references and rvalue references are distinct types. Except where explicitly noted, they are semantically equivalent and commonly referred to as references.

1 [*Example:*

```
void f(double& a) { a += 3.14; }
// ...
double d = 0;
f(d);
```

declares `a` to be a reference parameter of `f` so the call `f(d)` will add `3.14` to `d`.

```
int v[20];
// ...
int& g(int i) { return v[i]; }
// ...
g(3) = 7;
```

declares the function `g()` to return a reference to an integer so `g(3)=7` will assign `7` to the fourth element of the array `v`. For another example,

```
struct link {
  link* next;
};

link* first;

void h(link*& p) {  // p is a reference to pointer
  p->next = first;
  first = p;
  p = 0;
}

void k() {
   link* q = new link;
   h(q);
}
```

declares `p` to be a reference to a pointer to `link` so `h(q)` will leave `q` with the value zero. See also 8.2156.1. — *end example*]

2

It is unspecified whether or not a reference requires storage (3.2235).

1

There shall be no references to references, no arrays of references, and no pointers to references. The declaration of a reference shall contain an *initializer* (8.2156.1) except when the declaration contains an explicit `extern` specifier (7.49.1), is a class member (9.194) declaration within a class definition, or is the declaration of a parameter or a return type (8.360.1); see 3.43. A reference shall be initialized to refer to a valid object or function. [*Note:* in particular, a null reference cannot exist in a well-defined program, because the only way to create such a reference would be to bind it to the "object" obtained by indirection through a null pointer, which causes undefined behavior. As described in 9.921, a reference cannot be bound directly to a bit-field. — *end note*]

1

If a *typedef-name* (7.235.1, 14.97) or a *decltype-specifier* (7.678.0.1) denotes a type `TR` that is a reference to a type `T`, an attempt to create the type "lvalue reference to *cv* `TR`" creates the type "lvalue reference to `T`", while an attempt to create the type "rvalue reference to *cv* `TR`" creates the type `TR`. [*Example:*

```
int i;
typedef int& LRI;
typedef int&& RRI;

LRI& r1 = i;                    // r1 has the type int&
const LRI& r2 = i;              // r2 has the type int&
const LRI&& r3 = i;             // r3 has the type int&

RRI& r4 = i;                    // r4 has the type int&
RRI&& r5 = 5;                   // r5 has the type int&&

decltype(r2)& r6 = i;           // r6 has the type int&
decltype(r2)&& r7 = i;          // r7 has the type int&
```

— *end example*]

1 [*Note:* Forming a reference to function type is ill-formed if the function type has *cv-qualifier*s or a *ref-qualifier*; see 8.360.1. — *end note*]

8.236.1 Pointers to members [dcl.mptr]

1

In a declaration `T D` where `D` has the form

nested-name-specifier `*` *attribute-specifier-seq*$_{opt}$ *cv-qualifier-seq*$_{opt}$ *D1*

and the *nested-name-specifier* denotes a class, and the type of the identifier in the declaration `T D1` is "*derived-declarator-type-list* `T`", then the type of the identifier of `D` is "*derived-declarator-type-list cv-qualifier-seq* pointer to member of class *nested-name-specifier* of type `T`". The optional *attribute-specifier-seq* (7.1546.1) appertains to the pointer-to-member.

1 [*Example:*

```
struct X {
  void f(int);
  int a;
};
struct Y;
```

```
int X::* pmi = &X::a;
void (X::* pmf)(int) = &X::f;
double X::* pmd;
char Y::* pmc;
```

declares `pmi`, `pmf`, `pmd` and `pmc` to be a pointer to a member of `X` of type `int`, a pointer to a member of `X` of type `void(int)`, a pointer to a member of `X` of type `double` and a pointer to a member of `Y` of type `char` respectively. The declaration of `pmd` is well-formed even though `X` has no members of type `double`. Similarly, the declaration of `pmc` is well-formed even though `Y` is an incomplete type. `pmi` and `pmf` can be used like this:

```
X obj;
// ...
obj.*pmi = 7;        // assign 7 to an integer
                     // member of obj
(obj.*pmf)(7);       // call a function member of obj
                     // with the argument 7
```

— *end example*]

2

A pointer to member shall not point to a static member of a class (9.559), a member with reference type, or "*cv* `void`."

[*Note:* See also 5.1897 and 5.2832. The type "pointer to member" is distinct from the type "pointer", that is, a pointer to member is declared only by the pointer to member declarator syntax, and never by the pointer declarator syntax. There is no "reference-to-member" type in C++. — *end note*]

8.251.1 Arrays [dcl.array]

1

In a declaration `T D` where `D` has the form

`D1 [` *constant-expression*$_{opt}$`]` *attribute-specifier-seq*$_{opt}$

and the type of the identifier in the declaration `T D1` is "*derived-declarator-type-list* `T`", then the type of the identifier of `D` is an array type; if the type of the identifier of `D` contains the `auto` *type-specifier*, the program is ill-formed. `T` is called the array *element type*; this type shall not be a reference type, the (possibly cv-qualified) type `void`, a function type or an abstract class type. If the *constant-expression* (5.3643) is present, it shall be a converted constant expression of type `std::size_t` and its value shall be greater than zero. The constant expression specifies the *bound* of (number of elements in) the array. If the value of the constant expression is `N`, the array has `N` elements numbered `0` to `N-1`, and the type of the identifier of `D` is "*derived-declarator-type-list* array of `N T`". An object of array type contains a contiguously allocated non-empty set of `N` subobjects of type `T`. Except as noted below, if the constant expression is omitted, the type of the identifier of `D` is "*derived-declarator-type-list* array of unknown bound of `T`", an incomplete object type. The type "*derived-declarator-type-list* array of `N T`" is a different type from the type "*derived-declarator-type-list* array of unknown bound of `T`", see 3.2955. Any type of the form "*cv-qualifier-seq* array of `N T`" is adjusted to "array of `N` *cv-qualifier-seq* `T`", and similarly for "array of unknown bound of `T`". The optional *attribute-specifier-seq* appertains to the array. [*Example:*

```
typedef int A[5], AA[2][3];
typedef const A CA;          // type is "array of 5 const int"
typedef const AA CAA;        // type is "array of 2 array of 3 const int"
```

— *end example*] [*Note:* An "array of `N` *cv-qualifier-seq* `T`" has cv-qualified type; see 3.3640.1. — *end note*]

1

An array can be constructed from one of the fundamental types (except `void`), from a pointer, from a pointer to member, from a class, from an enumeration type, or from another array.

1

When several "array of" specifications are adjacent, a multidimensional array is created; only the first of the constant expressions that specify the bounds of the arrays may be omitted. In addition to declarations in which an incomplete object type is allowed, an array bound may be omitted in some cases in the declaration of a function parameter (8.360.1). An array bound may also be omitted when the declarator is followed by an *initializer* (8.821). In this case the bound is calculated from the number of initial elements (say, `N`) supplied (8.1817.1), and the type of the identifier of `D` is "array of `N` `T`." Furthermore, if there is a preceding declaration of the entity in the same scope in which the bound was specified, an omitted array bound is taken to be the same as in that earlier declaration, and similarly for the definition of a static data member of a class.

1 [*Example:*

```
float fa[17], *afp[17];
```

declares an array of `float` numbers and an array of pointers to `float` numbers. For another example,

```
static int x3d[3][5][7];
```

declares a static three-dimensional array of integers, with rank $3 \times 5 \times 7$. In complete detail, `x3d` is an array of three items; each item is an array of five arrays; each of the latter arrays is an array of seven integers. Any of the expressions `x3d`, `x3d[i]`, `x3d[i][j]`, `x3d[i][j][k]` can reasonably appear in an expression. Finally,

```
extern int x[10];
struct S {
  static int y[10];
};

int x[];                    // OK: bound is 10
int S::y[];                 // OK: bound is 10

void f() {
  extern int x[];
  int i = sizeof(x);        // error: incomplete object type
}
```

— *end example*]

2

[*Note:* conversions affecting expressions of array type are described in 4.180. Objects of array types cannot be modified, see 3.3673. — *end note*]

1

[*Note:* Except where it has been declared for a class (13.1210.1), the subscript operator `[]` is interpreted in such a way that `E1[E2]` is identical to `*((E1)+(E2))`. Because of the conversion rules that apply to `+`, if `E1` is an array and `E2` an integer, then `E1[E2]` refers to the `E2`-th member of `E1`. Therefore, despite its asymmetric appearance, subscripting is a commutative operation.

1

A consistent rule is followed for multidimensional arrays. If `E` is an n-dimensional array of rank $i \times j \times \ldots \times k$, then `E` appearing in an expression that is subject to the array-to-pointer conversion (4.180) is converted to a pointer to an $(n-1)$-dimensional array with rank $j \times \ldots \times k$. If the `*` operator, either explicitly or implicitly

as a result of subscripting, is applied to this pointer, the result is the pointed-to $(n-1)$-dimensional array, which itself is immediately converted into a pointer.

1

[*Example:* consider

```
int x[3][5];
```

Here `x` is a 3×5 array of integers. When `x` appears in an expression, it is converted to a pointer to (the first of three) five-membered arrays of integers. In the expression `x[i]` which is equivalent to `*(x+i)`, `x` is first converted to a pointer as described; then `x+i` is converted to the type of `x`, which involves multiplying `i` by the length of the object to which the pointer points, namely five integer objects. The results are added and indirection applied to yield an array (of five integers), which in turn is converted to a pointer to the first of the integers. If there is another subscript the same argument applies again; this time the result is an integer. — *end example*] — *end note*]

1

[*Note:* It follows from all this that arrays in C++ are stored row-wise (last subscript varies fastest) and that the first subscript in the declaration helps determine the amount of storage consumed by an array but plays no other part in subscript calculations. — *end note*]

8.360.1 Functions [dcl.fct]

1

In a declaration `T D` where `D` has the form

`D1 (` *parameter-declaration-clause* `)` *cv-qualifier-seq*$_{opt}$
ref-qualifier$_{opt}$ *exception-specification*$_{opt}$ *attribute-specifier-seq*$_{opt}$

and the type of the contained *declarator-id* in the declaration `T D1` is "*derived-declarator-type-list* `T`", the type of the *declarator-id* in `D` is "*derived-declarator-type-list* function of (*parameter-declaration-clause*) *cv-qualifier-seq*$_{opt}$ *ref-qualifier*$_{opt}$ returning `T`". The optional *attribute-specifier-seq* appertains to the function type.

1

In a declaration `T D` where `D` has the form

`D1 (` *parameter-declaration-clause* `)` *cv-qualifier-seq*$_{opt}$
ref-qualifier$_{opt}$ *exception-specification*$_{opt}$ *attribute-specifier-seq*$_{opt}$ *trailing-return-type*

and the type of the contained *declarator-id* in the declaration `T D1` is "*derived-declarator-type-list* `T`", `T` shall be the single *type-specifier* `auto`. The type of the *declarator-id* in `D` is "*derived-declarator-type-list* function of (*parameter-declaration-clause*) *cv-qualifier-seq*$_{opt}$ *ref-qualifier*$_{opt}$ returning *trailing-return-type*". The optional *attribute-specifier-seq* appertains to the function type.

1

A type of either form is a *function type*.[100]

parameter-declaration-clause:
parameter-declaration-list$_{opt}$ ...$_{opt}$
parameter-declaration-list `,` ...

parameter-declaration-list:
parameter-declaration
parameter-declaration-list `,` *parameter-declaration*

100) As indicated by syntax, cv-qualifiers are a significant component in function return types.

parameter-declaration:
attribute-specifier-seq$_{opt}$ *decl-specifier-seq declarator*
attribute-specifier-seq$_{opt}$ *decl-specifier-seq declarator* `=` *initializer-clause*
attribute-specifier-seq$_{opt}$ *decl-specifier-seq abstract-declarator*$_{opt}$
attribute-specifier-seq$_{opt}$ *decl-specifier-seq abstract-declarator*$_{opt}$`=` *initializer-clause*

The optional *attribute-specifier-seq* in a *parameter-declaration* appertains to the parameter.

1

The *parameter-declaration-clause* determines the arguments that can be specified, and their processing, when the function is called. [*Note:* the *parameter-declaration-clause* is used to convert the arguments specified on the function call; see 5.955.1. — *end note*] If the *parameter-declaration-clause* is empty, the function takes no arguments. A parameter list consisting of a single unnamed parameter of non-dependent type `void` is equivalent to an empty parameter list. Except for this special case, a parameter shall not have type *cv* `void`. If the *parameter-declaration-clause* terminates with an ellipsis or a function parameter pack (14.703.1), the number of arguments shall be equal to or greater than the number of parameters that do not have a default argument and are not function parameter packs. Where syntactically correct and where "*...*" is not part of an *abstract-declarator*, "*, ...*" is synonymous with "*...*". [*Example:* the declaration

```
int printf(const char*, ...);
```

declares a function that can be called with varying numbers and types of arguments.

```
printf("hello world");
printf("a=%d b=%d", a, b);
```

However, the first argument must be of a type that can be converted to a `const char*` — *end example*] [*Note:* The standard header `<cstdarg>` contains a mechanism for accessing arguments passed using the ellipsis (see 5.955.1 and 18.295). — *end note*]

1

A single name can be used for several different functions in a single scope; this is function overloading (Clause 13). All declarations for a function shall agree exactly in both the return type and the parameter-type-list. The type of a function is determined using the following rules. The type of each parameter (including function parameter packs) is determined from its own *decl-specifier-seq* and *declarator*. After determining the type of each parameter, any parameter of type "array of `T`" or "function returning `T`" is adjusted to be "pointer to `T`" or "pointer to function returning `T`," respectively. After producing the list of parameter types, any top-level *cv-qualifier*s modifying a parameter type are deleted when forming the function type. The resulting list of transformed parameter types and the presence or absence of the ellipsis or a function parameter pack is the function's *parameter-type-list*. [*Note:* This transformation does not affect the types of the parameters. For example, `int(*)(const int p, decltype(p)*)` and `int(*)(int, const int*)` are identical types. — *end note*]

1

A function type with a *cv-qualifier-seq* or a *ref-qualifier* (including a type named by *typedef-name* (7.235.1, 14.97)) shall appear only as:

— the function type for a non-static member function,

— the function type to which a pointer to member refers,

— the top-level function type of a function typedef declaration or *alias-declaration*,

— the *type-id* in the default argument of a *type-parameter* (14.97), or

— the *type-id* of a *template-argument* for a *type-parameter* (14.349.1).

[*Example:*

```
typedef int FIC(int) const;
FIC f;                // ill-formed: does not declare a member function
struct S {
  FIC f;              // OK
};
FIC S::*pm = &S::f;   // OK
```

— *end example*]

The effect of a *cv-qualifier-seq* in a function declarator is not the same as adding cv-qualification on top of the function type. In the latter case, the cv-qualifiers are ignored. [*Note:* a function type that has a *cv-qualifier-seq* is not a cv-qualified type; there are no cv-qualified function types. — *end note*] [*Example:*

```
typedef void F();
struct S {
  const F f;          // OK: equivalent to: void f();
};
```

— *end example*] The return type, the parameter-type-list, the *ref-qualifier*, and the *cv-qualifier-seq*, but not the default arguments (8.545.1) or the exception specification (15.228), are part of the function type. [*Note:* Function types are checked during the assignments and initializations of pointers to functions, references to functions, and pointers to member functions. — *end note*]

1 [*Example:* the declaration

```
int fseek(FILE*, long, int);
```

declares a function taking three arguments of the specified types, and returning `int` (7.519.1). — *end example*]

2

If the type of a parameter includes a type of the form "pointer to array of unknown bound of `T`" or "reference to array of unknown bound of `T`," the program is ill-formed.[101] Functions shall not have a return type of type array or function, although they may have a return type of type pointer or reference to such things. There shall be no arrays of functions, although there can be arrays of pointers to functions.

1

Types shall not be defined in return or parameter types. The type of a parameter or the return type for a function definition shall not be an incomplete class type (possibly cv-qualified) unless the function is deleted (8.793.1) or the definition is nested within the *member-specification* for that class (including definitions in nested classes defined within the class).

1

A typedef of function type may be used to declare a function but shall not be used to define a function (8.706). [*Example:*

```
typedef void F();
F  fv;                // OK: equivalent to void fv();
F  fv { }             // ill-formed
void fv() { }         // OK: definition of fv
```

— *end example*]

101) This excludes parameters of type "*ptr-arr-seq* `T2`" where `T2` is "pointer to array of unknown bound of `T`" and where *ptr-arr-seq* means any sequence of "pointer to" and "array of" derived declarator types. This exclusion applies to the parameters of the function, and if a parameter is a pointer to function or pointer to member function then to its parameters also, etc.

1

An identifier can optionally be provided as a parameter name; if present in a function definition (8.706), it names a parameter. [*Note:* In particular, parameter names are also optional in function definitions and names used for a parameter in different declarations and the definition of a function need not be the same. If a parameter name is present in a function declaration that is not a definition, it cannot be used outside of its function declarator because that is the extent of its potential scope (3.729.1). — *end note*]

1

[*Example:* the declaration

```
int i,
    *pi,
    f(),
    *fpi(int),
    (*pif)(const char*, const char*),
    (*fpif(int))(int);
```

declares an integer `i`, a pointer `pi` to an integer, a function `f` taking no arguments and returning an integer, a function `fpi` taking an integer argument and returning a pointer to an integer, a pointer `pif` to a function which takes two pointers to constant characters and returns an integer, a function `fpif` taking an integer argument and returning a pointer to a function that takes an integer argument and returns an integer. It is especially useful to compare `fpi` and `pif`. The binding of `*fpi(int)` is `*(fpi(int))`, so the declaration suggests, and the same construction in an expression requires, the calling of a function `fpi`, and then using indirection through the (pointer) result to yield an integer. In the declarator `(*pif)(const char*, const char*)`, the extra parentheses are necessary to indicate that indirection through a pointer to a function yields a function, which is then called. — *end example*] [*Note:* Typedefs and *trailing-return-type*s are sometimes convenient when the return type of a function is complex. For example, the function `fpif` above could have been declared

```
typedef int  IFUNC(int);
IFUNC*  fpif(int);
```

or

```
auto fpif(int)->int(*)(int);
```

A *trailing-return-type* is most useful for a type that would be more complicated to specify before the *declarator-id*:

```
template <class T, class U> auto add(T t, U u) -> decltype(t + u);
```

rather than

```
template <class T, class U> decltype((*(T*)0) + (*(U*)0)) add(T t, U u);
```

— *end note*]

1 A *non-template function* is a function that is not a function template specialization. [*Note:* A function template is not a function. — *end note*]

2 A *declarator-id* or *abstract-declarator* containing an ellipsis shall only be used in a *parameter-declaration*. Such a *parameter-declaration* is a parameter pack (14.703.1). When it is part of a *parameter-declaration-clause*, the parameter pack is a function parameter pack (14.703.1). [*Note:* Otherwise, the *parameter-declaration* is part of a *template-parameter-list* and the parameter pack is a template parameter pack; see 14.97. — *end note*] A function parameter pack is a pack expansion (14.703.1). [*Example:*

```
template<typename... T> void f(T (* ...t)(int, int));
```

```
int add(int, int);
float subtract(int, int);

void g() {
  f(add, subtract);
}
```

— *end example*]

3 There is a syntactic ambiguity when an ellipsis occurs at the end of a *parameter-declaration-clause* without a preceding comma. In this case, the ellipsis is parsed as part of the *abstract-declarator* if the type of the parameter either names a template parameter pack that has not been expanded or contains `auto`; otherwise, it is parsed as part of the *parameter-declaration-clause*.[102]

8.545.1 Default arguments [dcl.fct.default]

1

If an *initializer-clause* is specified in a *parameter-declaration* this *initializer-clause* is used as a default argument. Default arguments will be used in calls where trailing arguments are missing.

1

[*Example:* the declaration

```
void point(int = 3, int = 4);
```

declares a function that can be called with zero, one, or two arguments of type `int`. It can be called in any of these ways:

```
point(1,2);  point(1);  point();
```

The last two calls are equivalent to `point(1,4)` and `point(3,4)`, respectively. — *end example*]

1

A default argument shall be specified only in the *parameter-declaration-clause* of a function declaration or in a *template-parameter* (14.97); in the latter case, the *initializer-clause* shall be an *assignment-expression*. A default argument shall not be specified for a parameter pack. If it is specified in a *parameter-declaration-clause*, it shall not occur within a *declarator* or *abstract-declarator* of a *parameter-declaration*.[103]

1

For non-template functions, default arguments can be added in later declarations of a function in the same scope. Declarations in different scopes have completely distinct sets of default arguments. That is, declarations in inner scopes do not acquire default arguments from declarations in outer scopes, and vice versa. In a given function declaration, each parameter subsequent to a parameter with a default argument shall have a default argument supplied in this or a previous declaration or shall be a function parameter pack. A default argument shall not be redefined by a later declaration (not even to the same value). [*Example:*

```
void g(int = 0, ...);          // OK, ellipsis is not a parameter so it can follow
                               // a parameter with a default argument
void f(int, int);
void f(int, int = 7);
```

102) One can explicitly disambiguate the parse either by introducing a comma (so the ellipsis will be parsed as part of the *parameter-declaration-clause*) or by introducing a name for the parameter (so the ellipsis will be parsed as part of the *declarator-id*).

103) This means that default arguments cannot appear, for example, in declarations of pointers to functions, references to functions, or `typedef` declarations.

```
void h() {
  f(3);                          // OK, calls f(3, 7)
  void f(int = 1, int);          // error: does not use default
                                 // from surrounding scope
}
void m() {
  void f(int, int);              // has no defaults
  f(4);                          // error: wrong number of arguments
  void f(int, int = 5);          // OK
  f(4);                          // OK, calls f(4, 5);
  void f(int, int = 5);          // error: cannot redefine, even to
                                 // same value
}
void n() {
  f(6);                          // OK, calls f(6, 7)
}
```

— *end example*] For a given inline function defined in different translation units, the accumulated sets of default arguments at the end of the translation units shall be the same; see 3.91. If a friend declaration specifies a default argument expression, that declaration shall be a definition and shall be the only declaration of the function or function template in the translation unit.

1

The default argument has the same semantic constraints as the initializer in a declaration of a variable of the parameter type, using the copy-initialization semantics (8.821). The names in the default argument are bound, and the semantic constraints are checked, at the point where the default argument appears. Name lookup and checking of semantic constraints for default arguments in function templates and in member functions of class templates are performed as described in 14.1644.1. [*Example:* in the following code, `g` will be called with the value `f(2)`:

```
int a = 1;
int f(int);
int g(int x = f(a));             // default argument: f(::a)

void h() {
  a = 2;
  {
  int a = 3;
  g();                           // g(f(::a))
  }
}
```

— *end example*] [*Note:* In member function declarations, names in default arguments are looked up as described in 3.952.1. Access checking applies to names in default arguments as described in Clause 11. — *end note*]

1

Except for member functions of class templates, the default arguments in a member function definition that appears outside of the class definition are added to the set of default arguments provided by the member function declaration in the class definition; the program is ill-formed if a default constructor (12.30), copy or move constructor, or copy or move assignment operator (12.1577) is so declared. Default arguments for a member function of a class template shall be specified on the initial declaration of the member function within the class template. [*Example:*

```
class C {
```

```
  void f(int i = 3);
  void g(int i, int j = 99);
};

void C::f(int i = 3) {          // error: default argument already
}                               // specified in class scope
void C::g(int i = 88, int j) {  // in this translation unit,
}                               // C::g can be called with no argument
```

— *end example*]

1

Local variables shall not be used in a default argument. [*Example:*

```
void f() {
  int i;
  extern void g(int x = i);     //error
  // ...
}
```

— *end example*]

1

The keyword `this` shall not be used in a default argument of a member function. [*Example:*

```
class A {
  void f(A* p = this) { }       // error
};
```

— *end example*]

1

A default argument is evaluated each time the function is called with no argument for the corresponding parameter. The order of evaluation of function arguments is unspecified. Consequently, parameters of a function shall not be used in a default argument, even if they are not evaluated. Parameters of a function declared before a default argument are in scope and can hide namespace and class member names. [*Example:*

```
int a;
int f(int a, int b = a);            // error: parameter a
                                    // used as default argument
typedef int I;
int g(float I, int b = I(2));       // error: parameter I found
int h(int a, int b = sizeof(a));    // error, parameter a used
                                    // in default argument
```

— *end example*] Similarly, a non-static member shall not be used in a default argument, even if it is not evaluated, unless it appears as the *id-expression* of a class member access expression (5.1194.1) or unless it is used to form a pointer to member (5.1898.1). [*Example:* the declaration of `X::mem1()` in the following example is ill-formed because no object is supplied for the non-static member `X::a` used as an initializer.

```
int b;
class X {
  int a;
  int mem1(int i = a);          // error: non-static member a
                                // used as default argument
  int mem2(int i = b);          // OK; use X::b
  static int b;
```

```
};
```

The declaration of `X::mem2()` is meaningful, however, since no object is needed to access the static member `X::b`. Classes, objects, and members are described in Clause 9. — *end example*] A default argument is not part of the type of a function. [*Example:*

```
int f(int = 0);

void h() {
  int j = f(1);
  int k = f();                    // OK, means f(0)
}

int (*p1)(int) = &f;
int (*p2)() = &f;                 // error: type mismatch
```

— *end example*] When a declaration of a function is introduced by way of a *using-declaration* (7.1311.1), any default argument information associated with the declaration is made known as well. If the function is redeclared thereafter in the namespace with additional default arguments, the additional arguments are also known at any point following the redeclaration where the *using-declaration* is in scope.

1

A virtual function call (10.113) uses the default arguments in the declaration of the virtual function determined by the static type of the pointer or reference denoting the object. An overriding function in a derived class does not acquire default arguments from the function it overrides. [*Example:*

```
struct A {
  virtual void f(int a = 7);
};
struct B : public A {
  void f(int a);
};
void m() {
  B* pb = new B;
  A* pa = pb;
  pa->f();            // OK, calls pa->B::f(7)
  pb->f();            // error: wrong number of arguments for B::f()
}
```

— *end example*]

8.706 Function definitions [dcl.fct.def]

8.706.1 In general [dcl.fct.def.general]

1

Function definitions have the form

function-definition:
  attribute-specifier-seq$_{opt}$ *decl-specifier-seq*$_{opt}$ *declarator virt-specifier-seq*$_{opt}$ *function-body*

function-body:
  ctor-initializer$_{opt}$ *compound-statement*
  function-try-block
  `= default ;`
  `= delete ;`

Any informal reference to the body of a function should be interpreted as a reference to the non-terminal *function-body*. The optional *attribute-specifier-seq* in a *function-definition* appertains to the function. A *virt-specifier-seq* can be part of a *function-definition* only if it is a *member-declaration* (9.194).

1

The *declarator* in a *function-definition* shall have the form

`D1 (` *parameter-declaration-clause* `)` *cv-qualifier-seq*$_{opt}$ *ref-qualifier*$_{opt}$ *exception-specification*$_{opt}$ *attribute-specifier-seq*$_{opt}$ *trailing-return-type*$_{opt}$

as described in 8.360.1. A function shall be defined only in namespace or class scope.

1 [*Example:* a simple example of a complete function definition is

```
int max(int a, int b, int c) {
  int m = (a > b) ? a : b;
  return (m > c) ? m : c;
}
```

Here `int` is the *decl-specifier-seq*; `max(int a, int b, int c)` is the *declarator*; `{ /* ... */ }` is the *function-body*. — *end example*]

2 A *ctor-initializer* is used only in a constructor; see 12.30 and 12.1011.

3

A *cv-qualifier-seq* or a *ref-qualifier* (or both) can be part of a non-static member function declaration, non-static member function definition, or pointer to member function only (8.360.1); see 9.508.1.

1

[*Note:* Unused parameters need not be named. For example,

```
void print(int a, int) {
  std::printf("a = %d\n",a);
}
```

— *end note*]

1 In the *function-body*, a *function-local predefined variable* denotes a block-scope object of static storage duration that is implicitly defined (see 3.694.1).

2

The function-local predefined variable `__func__` is defined as if a definition of the form

```
static const char __func__[] = "function-name";
```

had been provided, where *function-name* is an implementation-defined string. It is unspecified whether such a variable has an address distinct from that of any other object in the program.[104]

[*Example:*

```
struct S {
  S() : s(__func__) { }          // OK
  const char* s;
};
void f(const char* s = __func__);  // error: __func__ is undeclared
```

— *end example*]

104) Implementations are permitted to provide additional predefined variables with names that are reserved to the implementation (17.332.0.0.1). If a predefined variable is not odr-used (3.91), its string value need not be present in the program image.

8.730.1 Explicitly-defaulted functions [dcl.fct.def.default]

1

A function definition of the form:

attribute-specifier-seq$_{opt}$ *decl-specifier-seq*$_{opt}$ *declarator virt-specifier-seq*$_{opt}$ `= default ;`

is called an *explicitly-defaulted* definition. A function that is explicitly defaulted shall

— be a special member function,

— have the same declared function type (except for possibly differing *ref-qualifier*s and except that in the case of a copy constructor or copy assignment operator, the parameter type may be "reference to non-const `T`", where `T` is the name of the member function's class) as if it had been implicitly declared, and

— not have default arguments.

1

An explicitly-defaulted function may be declared `constexpr` only if it would have been implicitly declared as `constexpr`. If a function is explicitly defaulted on its first declaration,

— it is implicitly considered to be `constexpr` if the implicit declaration would be, and,

— it is implicitly considered to have the same *exception-specification* as if it had been implicitly declared (15.228).

1

If a function that is explicitly defaulted has an explicit *exception-specification* that is not compatible (15.228) with the *exception-specification* on the implicit declaration, then

— if the function is explicitly defaulted on its first declaration, it is defined as deleted;

— otherwise, the program is ill-formed.

1 [*Example:*

```
struct S {
  constexpr S() = default;                  // ill-formed: implicit S() is not constexpr
  S(int a = 0) = default;                   // ill-formed: default argument
  void operator=(const S&) = default;       // ill-formed: non-matching return type
  ~S() throw(int) = default;                // deleted: exception specification does not match
private:
  int i;
  S(S&);                                    // OK: private copy constructor
};
S::S(S&) = default;                         // OK: defines copy constructor
```

— *end example*]

2

Explicitly-defaulted functions and implicitly-declared functions are collectively called *defaulted* functions, and the implementation shall provide implicit definitions for them (12.30 12.807, 12.1577), which might mean defining them as deleted. A function is *user-provided* if it is user-declared and not explicitly defaulted or deleted on its first declaration. A user-provided explicitly-defaulted function (i.e., explicitly defaulted after its first declaration) is defined at the point where it is explicitly defaulted; if such a function is implicitly defined as deleted, the program is ill-formed. [*Note:* Declaring a function as defaulted after its

first declaration can provide efficient execution and concise definition while enabling a stable binary interface to an evolving code base. — *end note*]

1 [*Example:*

```
struct trivial {
  trivial() = default;
  trivial(const trivial&) = default;
  trivial(trivial&&) = default;
  trivial& operator=(const trivial&) = default;
  trivial& operator=(trivial&&) = default;
  ~trivial() = default;
};

struct nontrivial1 {
  nontrivial1();
};
nontrivial1::nontrivial1() = default;          // not first declaration
```

— *end example*]

8.793.1 Deleted definitions [dcl.fct.def.delete]

1

A function definition of the form:

attribute-specifier-seq$_{opt}$ *decl-specifier-seq*$_{opt}$ *declarator virt-specifier-seq*$_{opt}$ `= delete ;`

is called a *deleted definition*. A function with a deleted definition is also called a *deleted function*.

1

A program that refers to a deleted function implicitly or explicitly, other than to declare it, is ill-formed. [*Note:* This includes calling the function implicitly or explicitly and forming a pointer or pointer-to-member to the function. It applies even for references in expressions that are not potentially-evaluated. If a function is overloaded, it is referenced only if the function is selected by overload resolution. — *end note*]

1

[*Example:* One can enforce non-default initialization and non-integral initialization with

```
struct onlydouble {
  onlydouble() = delete;              // OK, but redundant
  onlydouble(std::intmax_t) = delete;
  onlydouble(double);
};
```

— *end example*]

[*Example:* One can prevent use of a class in certain `new` expressions by using deleted definitions of a user-declared `operator new` for that class.

```
struct sometype {
  void* operator new(std::size_t) = delete;
  void* operator new[](std::size_t) = delete;
};
sometype* p = new sometype;     // error, deleted class operator new
sometype* q = new sometype[3];  // error, deleted class operator new[]
```

— *end example*]

[*Example:* One can make a class uncopyable, i.e. move-only, by using deleted definitions of the copy constructor and copy assignment operator, and then providing defaulted definitions of the move constructor and move assignment operator.

```
struct moveonly {
  moveonly() = default;
  moveonly(const moveonly&) = delete;
  moveonly(moveonly&&) = default;
  moveonly& operator=(const moveonly&) = delete;
  moveonly& operator=(moveonly&&) = default;
  ~moveonly() = default;
};
moveonly* p;
moveonly q(*p); // error, deleted copy constructor
```

— *end example*]

1

A deleted function is implicitly inline. [*Note:* The one-definition rule (3.91) applies to deleted definitions. — *end note*] A deleted definition of a function shall be the first declaration of the function or, for an explicit specialization of a function template, the first declaration of that specialization. [*Example:*

```
struct sometype {
  sometype();
};
sometype::sometype() = delete;        // ill-formed; not first declaration
```

— *end example*]

8.821 Initializers [dcl.init]

1

A declarator can specify an initial value for the identifier being declared. The identifier designates a variable being initialized. The process of initialization described in the remainder of 8.821 applies also to initializations specified by other syntactic contexts, such as the initialization of function parameters with argument expressions (5.955.1) or the initialization of return values (6.270.2).

initializer:
  brace-or-equal-initializer
  `(` *expression-list* `)`

brace-or-equal-initializer:
  `=` *initializer-clause*
  braced-init-list

initializer-clause:
  assignment-expression
  braced-init-list

initializer-list:
  initializer-clause `...`$_{opt}$
  initializer-list `,` *initializer-clause* `...`$_{opt}$

braced-init-list:
  `{` *initializer-list* `,`$_{opt}$ `}`
  `{ }`

1

Except for objects declared with the `constexpr` specifier, for which see 7.328.1, an *initializer* in the definition of a variable can consist of arbitrary expressions involving literals and previously declared variables and functions, regardless of the variable's storage duration. [*Example:*

```
int f(int);
int a = 2;
int b = f(a);
int c(b);
```

— *end example*]

1 [*Note:* Default arguments are more restricted; see 8.545.1.

2 The order of initialization of variables with static storage duration is described in 3.1721 and 6.419. — *end note*]

3

A declaration of a block-scope variable with external or internal linkage that has an *initializer* is ill-formed.

1

To *zero-initialize* an object or reference of type `T` means:

— if `T` is a scalar type (3.2955), the object is initialized to the value obtained by converting the integer literal `0` (zero) to `T`;[105]

— if `T` is a (possibly cv-qualified) non-union class type, each non-static data member and each base-class subobject is zero-initialized and padding is initialized to zero bits;

— if `T` is a (possibly cv-qualified) union type, the object's first non-static named data member is zero-initialized and padding is initialized to zero bits;

— if `T` is an array type, each element is zero-initialized;

— if `T` is a reference type, no initialization is performed.

1

To *default-initialize* an object of type `T` means:

— if `T` is a (possibly cv-qualified) class type (Clause 9), the default constructor (12.30) for `T` is called (and the initialization is ill-formed if `T` has no default constructor or overload resolution (13.88) results in an ambiguity or in a function that is deleted or inaccessible from the context of the initialization);

— if `T` is an array type, each element is default-initialized;

— otherwise, no initialization is performed.

If a program calls for the default initialization of an object of a const-qualified type `T`, `T` shall be a class type with a user-provided default constructor.

1

To *value-initialize* an object of type `T` means:

— if `T` is a (possibly cv-qualified) class type (Clause 9) with either no default constructor (12.30) or a default constructor that is user-provided or deleted, then the object is default-initialized;

— if `T` is a (possibly cv-qualified) class type without a user-provided or deleted default constructor, then the object is zero-initialized and the semantic constraints for default-initialization are checked, and if `T` has a non-trivial default constructor, the object is default-initialized;

105) As specified in 4.579, converting an integer literal whose value is `0` to a pointer type results in a null pointer value.

— if `T` is an array type, then each element is value-initialized;

— otherwise, the object is zero-initialized.

An object that is value-initialized is deemed to be constructed and thus subject to provisions of this International Standard applying to "constructed" objects, objects "for which the constructor has completed," etc., even if no constructor is invoked for the object's initialization.

1

A program that calls for default-initialization or value-initialization of an entity of reference type is ill-formed.

1

[*Note:* Every object of static storage duration is zero-initialized at program startup before any other initialization takes place. In some cases, additional initialization is done later. — *end note*]

1

An object whose initializer is an empty set of parentheses, i.e., `()`, shall be value-initialized.

[*Note:* Since `()` is not permitted by the syntax for *initializer*,

```
X a();
```

is not the declaration of an object of class `X`, but the declaration of a function taking no argument and returning an `X`. The form `()` is permitted in certain other initialization contexts (5.2221.1, 5.1108.1, 12.1026.1). — *end note*]

1

If no initializer is specified for an object, the object is default-initialized. When storage for an object with automatic or dynamic storage duration is obtained, the object has an *indeterminate value*, and if no initialization is performed for the object, that object retains an indeterminate value until that value is replaced (5.3483). [*Note:* Objects with static or thread storage duration are zero-initialized, see 3.1937.1. — *end note*] If an indeterminate value is produced by an evaluation, the behavior is undefined except in the following cases:

— If an indeterminate value of unsigned narrow character type (3.3151.1) is produced by the evaluation of:

  — the second or third operand of a conditional expression (5.3337),

  — the right operand of a comma expression (5.3602),

  — the operand of a cast or conversion to an unsigned narrow character type (4.397, 5.1108.1, 5.1443.1, 5.2765), or

  — a discarded-value expression (Clause 5),

  then the result of the operation is an indeterminate value.

— If an indeterminate value of unsigned narrow character type is produced by the evaluation of the right operand of a simple assignment operator (5.3483) whose first operand is an lvalue of unsigned narrow character type, an indeterminate value replaces the value of the object referred to by the left operand.

— If an indeterminate value of unsigned narrow character type is produced by the evaluation of the initialization expression when initializing an object of unsigned narrow character type, that object is initialized to an indeterminate value.

[*Example:*

```
int f(bool b) {
  unsigned char c;
  unsigned char d = c; // OK, d has an indeterminate value
  int e = d;           // undefined behavior
  return b ? d : 0;    // undefined behavior if b is true
}
```

— *end example*]

1

An initializer for a static member is in the scope of the member's class. [*Example:*

```
int a;

struct X {
  static int a;
  static int b;
};

int X::a = 1;
int X::b = a;       // X::b = X::a
```

— *end example*]

1

The form of initialization (using parentheses or `=`) is generally insignificant, but does matter when the initializer or the entity being initialized has a class type; see below. If the entity being initialized does not have class type, the *expression-list* in a parenthesized initializer shall be a single expression.

1

The initialization that occurs in the form

```
T x = a;
```

as well as in argument passing, function return, throwing an exception (15.7), handling an exception (15.146), and aggregate member initialization (8.1817.1) is called *copy-initialization*. [*Note:* Copy-initialization may invoke a move (12.1577). — *end note*]

1

The initialization that occurs in the forms

```
T x(a);
T x{a};
```

as well as in `new` expressions (5.2221.1), `static_cast` expressions (5.1443.1), functional notation type conversions (5.1108.1), and base and member initializers (12.1026.1) is called *direct-initialization*.

1

The semantics of initializers are as follows. The *destination type* is the type of the object or reference being initialized and the *source type* is the type of the initializer expression. If the initializer is not a single (possibly parenthesized) expression, the source type is not defined.

— If the initializer is a (non-parenthesized) *braced-init-list*, the object or reference is list-initialized (8.2435.1).

— If the destination type is a reference type, see 8.2156.1.

— If the destination type is an array of characters, an array of `char16_t`, an array of `char32_t`, or an array of `wchar_t`, and the initializer is a string literal, see 8.2115.1.

— If the initializer is `()`, the object is value-initialized.

— Otherwise, if the destination type is an array, the program is ill-formed.

— If the destination type is a (possibly cv-qualified) class type:

  — If the initialization is direct-initialization, or if it is copy-initialization where the cv-unqualified version of the source type is the same class as, or a derived class of, the class of the destination, constructors are considered. The applicable constructors are enumerated (13.354.0.1), and the best one is chosen through overload resolution (13.88). The constructor so selected is called to initialize the object, with the initializer expression or *expression-list* as its argument(s). If no constructor applies, or the overload resolution is ambiguous, the initialization is ill-formed.

  — Otherwise (i.e., for the remaining copy-initialization cases), user-defined conversion sequences that can convert from the source type to the destination type or (when a conversion function is used) to a derived class thereof are enumerated as described in 13.368.0.1, and the best one is chosen through overload resolution (13.88). If the conversion cannot be done or is ambiguous, the initialization is ill-formed. The function selected is called with the initializer expression as its argument; if the function is a constructor, the call initializes a temporary of the cv-unqualified version of the destination type. The temporary is a prvalue. The result of the call (which is the temporary for the constructor case) is then used to direct-initialize, according to the rules above, the object that is the destination of the copy-initialization. In certain cases, an implementation is permitted to eliminate the copying inherent in this direct-initialization by constructing the intermediate result directly into the object being initialized; see 12.306, 12.1577.

— Otherwise, if the source type is a (possibly cv-qualified) class type, conversion functions are considered. The applicable conversion functions are enumerated (13.396.0.1), and the best one is chosen through overload resolution (13.88). The user-defined conversion so selected is called to convert the initializer expression into the object being initialized. If the conversion cannot be done or is ambiguous, the initialization is ill-formed.

— Otherwise, the initial value of the object being initialized is the (possibly converted) value of the initializer expression. Standard conversions (Clause 4) will be used, if necessary, to convert the initializer expression to the cv-unqualified version of the destination type; no user-defined conversions are considered. If the conversion cannot be done, the initialization is ill-formed. [*Note:* An expression of type "*cv1* `T`" can initialize an object of type "*cv2* `T`" independently of the cv-qualifiers *cv1* and *cv2*.

```
int a;
const int b = a;
int c = b;
```

— *end note*]

1 An *initializer-clause* followed by an ellipsis is a pack expansion (14.703.1).

8.1817.1 Aggregates [dcl.init.aggr]

1

An *aggregate* is an array or a class (Clause 9) with no user-provided constructors (12.30), no private or protected non-static data members (Clause 11), no base classes (Clause 10), and no virtual functions (10.113).

1

When an aggregate is initialized by an initializer list, as specified in 8.2435.1, the elements of the initializer list are taken as initializers for the members of the aggregate, in increasing subscript or member order. Each

member is copy-initialized from the corresponding *initializer-clause*. If the *initializer-clause* is an expression and a narrowing conversion (8.2435.1) is required to convert the expression, the program is ill-formed. [*Note:* If an *initializer-clause* is itself an initializer list, the member is list-initialized, which will result in a recursive application of the rules in this section if the member is an aggregate. — *end note*] [*Example:*

```
struct A {
  int x;
  struct B {
    int i;
    int j;
  } b;
} a = { 1, { 2, 3 } };
```

initializes `a.x` with 1, `a.b.i` with 2, `a.b.j` with 3. — *end example*]

1

An aggregate that is a class can also be initialized with a single expression not enclosed in braces, as described in 8.821.

1

An array of unknown size initialized with a brace-enclosed *initializer-list* containing `n` *initializer-clause*s, where `n` shall be greater than zero, is defined as having `n` elements (8.251.1). [*Example:*

```
int x[] = { 1, 3, 5 };
```

declares and initializes `x` as a one-dimensional array that has three elements since no size was specified and there are three initializers. — *end example*] An empty initializer list `{}` shall not be used as the *initializer-clause* for an array of unknown bound.[106]

1

Static data members and anonymous bit-fields are not considered members of the class for purposes of aggregate initialization. [*Example:*

```
struct A {
  int i;
  static int s;
  int j;
  int :17;
  int k;
} a = { 1, 2, 3 };
```

Here, the second initializer 2 initializes `a.j` and not the static data member `A::s`, and the third initializer 3 initializes `a.k` and not the anonymous bit-field before it. — *end example*]

1

An *initializer-list* is ill-formed if the number of *initializer-clause*s exceeds the number of members or elements to initialize. [*Example:*

```
char cv[4] = { 'a', 's', 'd', 'f', 0 };      // error
```

is ill-formed. — *end example*]

1

If there are fewer *initializer-clause*s in the list than there are members in the aggregate, then each member not explicitly initialized shall be initialized from its *brace-or-equal-initializer* or, if there is no *brace-or-equal-initializer*, from an empty initializer list (8.2435.1). [*Example:*

106) The syntax provides for empty *initializer-list*s, but nonetheless C++ does not have zero length arrays.

```
struct S { int a; const char* b; int c; int d = b[a]; };
S ss = { 1, "asdf" };
```

initializes `ss.a` with 1, `ss.b` with `"asdf"`, `ss.c` with the value of an expression of the form `int{}` (that is, `0`), and `ss.d` with the value of `ss.b[ss.a]` (that is, `'s'`), and in

```
struct X { int i, j, k = 42; };
X a[] = { 1, 2, 3, 4, 5, 6 };
X b[2] = { { 1, 2, 3 }, { 4, 5, 6 } };
```

`a` and `b` have the same value — *end example*]

1

If an aggregate class `C` contains a subaggregate member `m` that has no members for purposes of aggregate initialization, the *initializer-clause* for `m` shall not be omitted from an *initializer-list* for an object of type `C` unless the *initializer-clause*s for all members of `C` following `m` are also omitted. [*Example:*

```
struct S { } s;
struct A {
  S s1;
  int i1;
  S s2;
  int i2;
  S s3;
  int i3;
} a = {
  { },        // Required initialization
  0,
  s,          // Required initialization
  0
};            // Initialization not required for A::s3 because A::i3 is also not initialized
```

— *end example*]

1

If an incomplete or empty *initializer-list* leaves a member of reference type uninitialized, the program is ill-formed.

1

When initializing a multi-dimensional array, the *initializer-clause*s initialize the elements with the last (rightmost) index of the array varying the fastest (8.251.1). [*Example:*

```
int x[2][2] = { 3, 1, 4, 2 };
```

initializes `x[0][0]` to `3`, `x[0][1]` to `1`, `x[1][0]` to `4`, and `x[1][1]` to `2`. On the other hand,

```
float y[4][3] = {
  { 1 }, { 2 }, { 3 }, { 4 }
};
```

initializes the first column of `y` (regarded as a two-dimensional array) and leaves the rest zero. — *end example*]

1

Braces can be elided in an *initializer-list* as follows. If the *initializer-list* begins with a left brace, then the succeeding comma-separated list of *initializer-clause*s initializes the members of a subaggregate; it is

erroneous for there to be more *initializer-clause*s than members. If, however, the *initializer-list* for a subaggregate does not begin with a left brace, then only enough *initializer-clause*s from the list are taken to initialize the members of the subaggregate; any remaining *initializer-clause*s are left to initialize the next member of the aggregate of which the current subaggregate is a member. [*Example:*

```
float y[4][3] = {
  { 1, 3, 5 },
  { 2, 4, 6 },
  { 3, 5, 7 },
};
```

is a completely-braced initialization: 1, 3, and 5 initialize the first row of the array `y[0]`, namely `y[0][0]`, `y[0][1]`, and `y[0][2]`. Likewise the next two lines initialize `y[1]` and `y[2]`. The initializer ends early and therefore `y[3]`s elements are initialized as if explicitly initialized with an expression of the form `float()`, that is, are initialized with `0.0`. In the following example, braces in the *initializer-list* are elided; however the *initializer-list* has the same effect as the completely-braced *initializer-list* of the above example,

```
float y[4][3] = {
  1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

The initializer for `y` begins with a left brace, but the one for `y[0]` does not, therefore three elements from the list are used. Likewise the next three are taken successively for `y[1]` and `y[2]`. — *end example*]

1

All implicit type conversions (Clause 4) are considered when initializing the aggregate member with an *assignment-expression*. If the *assignment-expression* can initialize a member, the member is initialized. Otherwise, if the member is itself a subaggregate, brace elision is assumed and the *assignment-expression* is considered for the initialization of the first member of the subaggregate. [*Note:* As specified above, brace elision cannot apply to subaggregates with no members for purposes of aggregate initialization; an *initializer-clause* for the entire subobject is required. — *end note*]

[*Example:*

```
struct A {
  int i;
  operator int();
};
struct B {
  A a1, a2;
  int z;
};
A a;
B b = { 4, a, a };
```

Braces are elided around the *initializer-clause* for `b.a1.i`. `b.a1.i` is initialized with 4, `b.a2` is initialized with `a`, `b.z` is initialized with whatever `a.operator int()` returns. — *end example*]

1

[*Note:* An aggregate array or an aggregate class may contain members of a class type with a user-provided constructor (12.30). Initialization of these aggregate objects is described in 12.1021.1. — *end note*]

1 [*Note:* Whether the initialization of aggregates with static storage duration is static or dynamic is specified in 3.1937.1 and 6.419. — *end note*]

2

When a union is initialized with a brace-enclosed initializer, the braces shall only contain an *initializer-clause* for the first non-static data member of the union. [*Example:*

```
union u { int a; const char* b; };
u a = { 1 };
u b = a;
u c = 1;                        // error
u d = { 0, "asdf" };            // error
u e = { "asdf" };               // error
```

— *end example*]

1 [*Note:* As described above, the braces around the *initializer-clause* for a union member can be omitted if the union is a member of another aggregate. — *end note*]

8.2115.1 Character arrays [dcl.init.string]

1

An array of narrow character type (3.3151.1), `char16_t` array, `char32_t` array, or `wchar_t` array can be initialized by a narrow string literal, `char16_t` string literal, `char32_t` string literal, or wide string literal, respectively, or by an appropriately-typed string literal enclosed in braces (2.453.1). Successive characters of the value of the string literal initialize the elements of the array. [*Example:*

```
char msg[] = "Syntax error on line %s\n";
```

shows a character array whose members are initialized with a *string-literal*. Note that because `'\n'` is a single character and because a trailing `'\0'` is appended, `sizeof(msg)` is `25`. — *end example*]

1

There shall not be more initializers than there are array elements. [*Example:*

```
char cv[4] = "asdf";            // error
```

is ill-formed since there is no space for the implied trailing `'\0'`. — *end example*]

1

If there are fewer initializers than there are array elements, each element not explicitly initialized shall be zero-initialized (8.821).

8.2156.1 References [dcl.init.ref]

1

A variable declared to be a `T&` or `T&&`, that is, "reference to type `T`" (8.97.1), shall be initialized by an object, or function, of type `T` or by an object that can be converted into a `T`. [*Example:*

```
int g(int);
void f() {
  int i;
  int& r = i;                   // r refers to i
  r = 1;                        // the value of i becomes 1
  int* p = &r;                  // p points to i
  int& rr = r;                  // rr refers to what r refers to, that is, to i
  int (&rg)(int) = g;           // rg refers to the function g
  rg(i);                        // calls function g
  int a[3];
  int (&ra)[3] = a;             // ra refers to the array a
  ra[1] = i;                    // modifies a[1]
}
```

— *end example*]

1

A reference cannot be changed to refer to another object after initialization. Note that initialization of a reference is treated very differently from assignment to it. Argument passing (5.955.1) and function value return (6.270.2) are initializations.

1

The initializer can be omitted for a reference only in a parameter declaration (8.360.1), in the declaration of a function return type, in the declaration of a class member within its class definition (9.194), and where the `extern` specifier is explicitly used. [*Example:*

```
int& r1;                    // error: initializer missing
extern int& r2;             // OK
```

— *end example*]

1

Given types "*cv1* `T1`" and "*cv2* `T2`," "*cv1* `T1`" is *reference-related* to "*cv2* `T2`" if `T1` is the same type as `T2`, or `T1` is a base class of `T2`. "*cv1* `T1`" is *reference-compatible* with "*cv2* `T2`" if `T1` is reference-related to `T2` and *cv1* is the same cv-qualification as, or greater cv-qualification than, *cv2*. In all cases where the reference-related or reference-compatible relationship of two types is used to establish the validity of a reference binding, and `T1` is a base class of `T2`, a program that necessitates such a binding is ill-formed if `T1` is an inaccessible (Clause 11) or ambiguous (10.52) base class of `T2`.

1

A reference to type "*cv1* `T1`" is initialized by an expression of type "*cv2* `T2`" as follows:

— If the reference is an lvalue reference and the initializer expression
  — is an lvalue (but is not a bit-field), and "*cv1* `T1`" is reference-compatible with "*cv2* `T2`," or
  — has a class type (i.e., `T2` is a class type), where `T1` is not reference-related to `T2`, and can be converted to an lvalue of type "*cv3* `T3`," where "*cv1* `T1`" is reference-compatible with "*cv3* `T3`"[107] (this conversion is selected by enumerating the applicable conversion functions (13.413.0.1) and choosing the best one through overload resolution (13.88)),

then the reference is bound to the initializer expression lvalue in the first case and to the lvalue result of the conversion in the second case (or, in either case, to the appropriate base class subobject of the object). [*Note:* The usual lvalue-to-rvalue (4.127), array-to-pointer (4.180), and function-to-pointer (4.220) standard conversions are not needed, and therefore are suppressed, when such direct bindings to lvalues are done. — *end note*]

[*Example:*

```
double d = 2.0;
double& rd = d;                 // rd refers to d
const double& rcd = d;          // rcd refers to d

struct A { };
struct B : A { operator int&(); } b;
A& ra = b;                      // ra refers to A subobject in b
const A& rca = b;               // rca refers to A subobject in b
int& ir = B();                  // ir refers to the result of B::operator int&
```

— *end example*]

107) This requires a conversion function (12.742.1) returning a reference type.

— Otherwise, the reference shall be an lvalue reference to a non-volatile const type (i.e., *cv1* shall be `const`), or the reference shall be an rvalue reference. [*Example:*

```
double& rd2 = 2.0;          // error: not an lvalue and reference not const
int  i = 2;
double& rd3 = i;            // error: type mismatch and reference not const
```

— *end example*]

— If the initializer expression

— is an xvalue (but not a bit-field), class prvalue, array prvalue or function lvalue and "*cv1* `T1`" is reference-compatible with "*cv2* `T2`", or

— has a class type (i.e., `T2` is a class type), where `T1` is not reference-related to `T2`, and can be converted to an xvalue, class prvalue, or function lvalue of type "*cv3* `T3`", where "*cv1* `T1`" is reference-compatible with "*cv3* `T3`" (see 13.413.0.1),

then the reference is bound to the value of the initializer expression in the first case and to the result of the conversion in the second case (or, in either case, to an appropriate base class subobject). In the second case, if the reference is an rvalue reference and the second standard conversion sequence of the user-defined conversion sequence includes an lvalue-to-rvalue conversion, the program is ill-formed.

[*Example:*

```
struct A { };
struct B : A { } b;
extern B f();
const A& rca2 = f();                // bound to the A subobject of the B rvalue.
A&& rra = f();                      // same as above
struct X {
  operator B();
  operator int&();
} x;
const A& r = x;                     // bound to the A subobject of the result of the conversion
int i2 = 42;
int&& rri = static_cast<int&&>(i2); // bound directly to i2
B&& rrb = x;                        // bound directly to the result of operator B
int&& rri2 = X();                   // error: lvalue-to-rvalue conversion applied to the
                                    // result of operator int&
```

— *end example*]

— Otherwise:

— If `T1` is a class type, user-defined conversions are considered using the rules for copy-initialization of an object of type "*cv1* `T1`" by user-defined conversion (8.821, 13.368.0.1); the program is ill-formed if the corresponding non-reference copy-initialization would be ill-formed. The result of the call to the conversion function, as described for the non-reference copy-initialization, is then used to direct-initialize the reference. The program is ill-formed if the direct-initialization does not result in a direct binding or if it involves a user-defined conversion.

— If `T1` is a non-class type, a temporary of type "*cv1* `T1`" is created and copy-initialized (8.821) from the initializer expression. The reference is then bound to the temporary.

If `T1` is reference-related to `T2`:

— *cv1* shall be the same cv-qualification as, or greater cv-qualification than, *cv2*; and

— if the reference is an rvalue reference, the initializer expression shall not be an lvalue.

[*Example:*

```
struct Banana { };
struct Enigma { operator const Banana(); };
void enigmatic() {
  typedef const Banana ConstBanana;
  Banana &&banana1 = ConstBanana(); // ill-formed
  Banana &&banana2 = Enigma();      // ill-formed
}

const double& rcd2 = 2;          // rcd2 refers to temporary with value 2.0
double&& rrd = 2;                // rrd refers to temporary with value 2.0
const volatile int cvi = 1;
const int& r2 = cvi;             // error: type qualifiers dropped
double d2 = 1.0;
double&& rrd2 = d2;              // error: copying lvalue of related type
int i3 = 2;
double&& rrd3 = i3;              // rrd3 refers to temporary with value 2.0
```

— *end example*]

In all cases except the last (i.e., creating and initializing a temporary from the initializer expression), the reference is said to *bind directly* to the initializer expression.

1 [*Note:* 12.306 describes the lifetime of temporaries bound to references. — *end note*]

8.2435.1 List-initialization [dcl.init.list]

1

List-initialization is initialization of an object or reference from a *braced-init-list*. Such an initializer is called an *initializer list*, and the comma-separated *initializer-clause*s of the list are called the *elements* of the initializer list. An initializer list may be empty. List-initialization can occur in direct-initialization or copy-initialization contexts; list-initialization in a direct-initialization context is called *direct-list-initialization* and list-initialization in a copy-initialization context is called *copy-list-initialization*. [*Note:* List-initialization can be used

— as the initializer in a variable definition (8.821)

— as the initializer in a new expression (5.2221.1)

— in a return statement (6.270.2)

— as a *for-range-initializer* (6.137)

— as a function argument (5.955.1)

— as a subscript (5.896.1)

— as an argument to a constructor invocation (8.821, 5.1108.1)

— as an initializer for a non-static data member (9.194)

— in a *mem-initializer* (12.1026.1)

— on the right-hand side of an assignment (5.3483)

[*Example:*

```
int a = {1};
std::complex<double> z{1,2};
new std::vector<std::string>{"once", "upon", "a", "time"};  // 4 string elements
f( {"Nicholas","Annemarie"} );  // pass list of two elements
return { "Norah" };             // return list of one element
int* e {};                      // initialization to zero / null pointer
x = double{1};                  // explicitly construct a double
std::map<std::string,int> anim = { {"bear",4}, {"cassowary",2}, {"tiger",7} };
```

— *end example*] — *end note*]

1

A constructor is an *initializer-list constructor* if its first parameter is of type `std::initializer_list<E>` or reference to possibly cv-qualified `std::initializer_list<E>` for some type `E`, and either there are no other parameters or else all other parameters have default arguments (8.545.1). [*Note:* Initializer-list constructors are favored over other constructors in list-initialization (13.425.0.1). Passing an initializer list as the argument to the constructor template `template<class T> C(T)` of a class `C` does not create an initializer-list constructor, because an initializer list argument causes the corresponding parameter to be a non-deduced context (14.2261.0.1). — *end note*] The template `std::initializer_list` is not predefined; if the header `<initializer_list>` is not included prior to a use of `std::initializer_list` — even an implicit use in which the type is not named (7.832.0.1) — the program is ill-formed.

1

List-initialization of an object or reference of type `T` is defined as follows:

— If `T` is an aggregate, aggregate initialization is performed (8.1817.1).

[*Example:*

```
double ad[] = { 1, 2.0 };          // OK
int ai[] = { 1, 2.0 };             // error: narrowing

struct S2 {
  int m1;
  double m2, m3;
};
S2 s21 = { 1, 2, 3.0 };            // OK
S2 s22 { 1.0, 2, 3 };              // error: narrowing
S2 s23 { };                        // OK: default to 0,0,0
```

— *end example*]

— Otherwise, if the initializer list has no elements and `T` is a class type with a default constructor, the object is value-initialized.

— Otherwise, if `T` is a specialization of `std::initializer_list<E>`, a prvalue `initializer_list` object is constructed as described below and used to initialize the object according to the rules for initialization of an object from a class of the same type (8.821).

— Otherwise, if `T` is a class type, constructors are considered. The applicable constructors are enumerated and the best one is chosen through overload resolution (13.88, 13.425.0.1). If a narrowing conversion (see below) is required to convert any of the arguments, the program is ill-formed.

[*Example:*

```
struct S {
  S(std::initializer_list<double>); // #1
  S(std::initializer_list<int>);    // #2
  S();                              // #3
  // ...
};
S s1 = { 1.0, 2.0, 3.0 };           // invoke #1
S s2 = { 1, 2, 3 };                 // invoke #2
S s3 = { };                         // invoke #3
```

— *end example*]

[*Example:*

```
struct Map {
  Map(std::initializer_list<std::pair<std::string,int>>);
};
Map ship = {{"Sophie",14}, {"Surprise",28}};
```

— *end example*]

[*Example:*

```
struct S {
  // no initializer-list constructors
  S(int, double, double);           // #1
  S();                              // #2
  // ...
};
S s1 = { 1, 2, 3.0 };               // OK: invoke #1
S s2 { 1.0, 2, 3 };                 // error: narrowing
S s3 { };                           // OK: invoke #2
```

— *end example*]

— Otherwise, if the initializer list has a single element of type `E` and either `T` is not a reference type or its referenced type is reference-related to `E`, the object or reference is initialized from that element; if a narrowing conversion (see below) is required to convert the element to `T`, the program is ill-formed.

[*Example:*

```
int x1 {2};                         // OK
int x2 {2.0};                       // error: narrowing
```

— *end example*]

— Otherwise, if `T` is a reference type, a prvalue temporary of the type referenced by `T` is copy-list-initialized or direct-list-initialized, depending on the kind of initialization for the reference, and the reference is bound to that temporary. [*Note:* As usual, the binding will fail and the program is ill-formed if the reference type is an lvalue reference to a non-const type. — *end note*]

[*Example:*

```
struct S {
  S(std::initializer_list<double>); // #1
  S(const std::string&);            // #2
  // ...
};
const S& r1 = { 1, 2, 3.0 };        // OK: invoke #1
```

```
const S& r2 { "Spinach" };          // OK: invoke #2
S& r3 = { 1, 2, 3 };                // error: initializer is not an lvalue
const int& i1 = { 1 };              // OK
const int& i2 = { 1.1 };            // error: narrowing
const int (&iar)[2] = { 1, 2 };     // OK: iar is bound to temporary array
```

— *end example*]

— Otherwise, if the initializer list has no elements, the object is value-initialized.

[*Example:*

```
int** pp {};                        // initialized to null pointer
```

— *end example*]

— Otherwise, the program is ill-formed.

[*Example:*

```
struct A { int i; int j; };
A a1 { 1, 2 };                      // aggregate initialization
A a2 { 1.2 };                       // error: narrowing
struct B {
  B(std::initializer_list<int>);
};
B b1 { 1, 2 };                      // creates initializer_list<int> and calls constructor
B b2 { 1, 2.0 };                    // error: narrowing
struct C {
  C(int i, double j);
};
C c1 = { 1, 2.2 };                  // calls constructor with arguments (1, 2.2)
C c2 = { 1.1, 2 };                  // error: narrowing

int j { 1 };                        // initialize to 1
int k { };                          // initialize to 0
```

— *end example*]

1

Within the *initializer-list* of a *braced-init-list*, the *initializer-clauses*, including any that result from pack expansions (14.703.1), are evaluated in the order in which they appear. That is, every value computation and side effect associated with a given *initializer-clause* is sequenced before every value computation and side effect associated with any *initializer-clause* that follows it in the comma-separated list of the *initializer-list*. [*Note:* This evaluation ordering holds regardless of the semantics of the initialization; for example, it applies when the elements of the *initializer-list* are interpreted as arguments of a constructor call, even though ordinarily there are no sequencing constraints on the arguments of a call. — *end note*]

1

An object of type `std::initializer_list<E>` is constructed from an initializer list as if the implementation allocated a temporary array of N elements of type `const E`, where N is the number of elements in the initializer list. Each element of that array is copy-initialized with the corresponding element of the initializer list, and the `std::initializer_list<E>` object is constructed to refer to that array. [*Note:* A constructor or conversion function selected for the copy shall be accessible (Clause 11) in the context of the initializer list. — *end note*] If a narrowing conversion is required to initialize any of the elements, the program is ill-formed.[*Example:*

```
struct X {
  X(std::initializer_list<double> v);
};
X x{ 1,2,3 };
```

The initialization will be implemented in a way roughly equivalent to this:

```
const double __a[3] = {double{1}, double{2}, double{3}};
X x(std::initializer_list<double>(__a, __a+3));
```

assuming that the implementation can construct an `initializer_list` object with a pair of pointers. — *end example*]

1

The array has the same lifetime as any other temporary object (12.306), except that initializing an `initializer_list` object from the array extends the lifetime of the array exactly like binding a reference to a temporary. [*Example:*

```
typedef std::complex<double> cmplx;
std::vector<cmplx> v1 = { 1, 2, 3 };

void f() {
  std::vector<cmplx> v2{ 1, 2, 3 };
  std::initializer_list<int> i3 = { 1, 2, 3 };
}

struct A {
  std::initializer_list<int> i4;
  A() : i4{ 1, 2, 3 } {}  // creates an A with a dangling reference
};
```

For `v1` and `v2`, the `initializer_list` object is a parameter in a function call, so the array created for `{ 1, 2, 3 }` has full-expression lifetime. For `i3`, the `initializer_list` object is a variable, so the array persists for the lifetime of the variable. For `i4`, the `initializer_list` object is initialized in a constructor's *ctor-initializer*, so the array persists only until the constructor exits, and so any use of the elements of `i4` after the constructor exits produces undefined behavior. — *end example*] [*Note:* The implementation is free to allocate the array in read-only memory if an explicit array with the same initializer could be so allocated. — *end note*]

1

A *narrowing conversion* is an implicit conversion

— from a floating-point type to an integer type, or

— from `long double` to `double` or `float`, or from `double` to `float`, except where the source is a constant expression and the actual value after conversion is within the range of values that can be represented (even if it cannot be represented exactly), or

— from an integer type or unscoped enumeration type to a floating-point type, except where the source is a constant expression and the actual value after conversion will fit into the target type and will produce the original value when converted back to the original type, or

— from an integer type or unscoped enumeration type to an integer type that cannot represent all the values of the original type, except where the source is a constant expression whose value after integral promotions will fit into the target type.

[*Note:* As indicated above, such conversions are not allowed at the top level in list-initializations. — *end note*] [*Example:*

```
int x = 999;                // x is not a constant expression
const int y = 999;
const int z = 99;
char c1 = x;                // OK, though it might narrow (in this case, it does narrow)
char c2{x};                 // error: might narrow
char c3{y};                 // error: narrows (assuming char is 8 bits)
char c4{z};                 // OK: no narrowing needed
unsigned char uc1 = {5};    // OK: no narrowing needed
unsigned char uc2 = {-1}; // error: narrows
unsigned int ui1 = {-1};    // error: narrows
signed int si1 =
  { (unsigned int)-1 };     // error: narrows
int ii = {2.0};             // error: narrows
float f1 { x };             // error: might narrow
float f2 { 7 };             // OK: 7 can be exactly represented as a float
int f(int);
int a[] =
  { 2, f(2), f(2.0) };      // OK: the double-to-int conversion is not at the top level
```

— *end example*]

9 Classes [class]

1

A class is a type. Its name becomes a *class-name* (9.172) within its scope.

class-name:
identifier
simple-template-id

*Class-specifier*s and *elaborated-type-specifier*s (7.808.0.1) are used to make *class-name*s. An object of a class consists of a (possibly empty) sequence of members and base class objects.

class-specifier:
class-head `{` *member-specification*$_{opt}$`}`

class-head:
class-key attribute-specifier-seq$_{opt}$ *class-head-name class-virt-specifier*$_{opt}$ *base-clause*$_{opt}$
class-key attribute-specifier-seq$_{opt}$ *base-clause*$_{opt}$

class-head-name:
nested-name-specifier$_{opt}$ *class-name*

class-virt-specifier:
`final`

class-key:
`class`
`struct`
`union`

A *class-specifier* whose *class-head* omits the *class-head-name* defines an unnamed class. [*Note:* An unnamed class thus can't be `final`. — *end note*]

1

A *class-name* is inserted into the scope in which it is declared immediately after the *class-name* is seen. The *class-name* is also inserted into the scope of the class itself; this is known as the *injected-class-name*. For purposes of access checking, the injected-class-name is treated as if it were a public member name. A *class-specifier* is commonly referred to as a class definition. A class is considered defined after the closing brace of its *class-specifier* has been seen even though its member functions are in general not yet defined. The optional *attribute-specifier-seq* appertains to the class; the attributes in the *attribute-specifier-seq* are thereafter considered attributes of the class whenever it is named.

1

If a class is marked with the *class-virt-specifier* `final` and it appears as a *base-type-specifier* in a *base-clause* (Clause 10), the program is ill-formed. Whenever a *class-key* is followed by a *class-head-name*, the *identifier* `final`, and a colon or left brace, `final` is interpreted as a *class-virt-specifier*. [*Example:*

```
struct A;
struct A final {};        // OK: definition of struct A,
                          // not value-initialization of variable final

struct X {
 struct C { constexpr operator int() { return 5; } };
 struct B final : C{};    // OK: definition of nested class B,
                          // not declaration of a bit-field member final
};
```

— *end example*]

1

Complete objects and member subobjects of class type shall have nonzero size.[108] [*Note:* Class objects can be assigned, passed as arguments to functions, and returned by functions (except objects of classes for which copying or moving has been restricted; see 12.1577). Other plausible operators, such as equality comparison, can be defined by the user; see 13.1122. — *end note*]

1

A *union* is a class defined with the *class-key* `union`; it holds at most one data member at a time (9.815). [*Note:* Aggregates of class type are described in 8.1817.1. — *end note*]

1

A *trivially copyable class* is a class that:

— has no non-trivial copy constructors (12.1577),

— has no non-trivial move constructors (12.1577),

— has no non-trivial copy assignment operators (13.1178.1, 12.1577),

— has no non-trivial move assignment operators (13.1178.1, 12.1577), and

— has a trivial destructor (12.807).

A *trivial class* is a class that has a default constructor (12.30), has no non-trivial default constructors, and is trivially copyable. [*Note:* In particular, a trivially copyable or trivial class does not have virtual functions or virtual base classes. — *end note*]

1

A *standard-layout class* is a class that:

— has no non-static data members of type non-standard-layout class (or array of such types) or reference,

— has no virtual functions (10.113) and no virtual base classes (10.31),

— has the same access control (Clause 11) for all non-static data members,

— has no non-standard-layout base classes,

— either has no non-static data members in the most derived class and at most one base class with non-static data members, or has no base classes with non-static data members, and

— has no base classes of the same type as the first non-static data member.[109]

1

A *standard-layout struct* is a standard-layout class defined with the *class-key* `struct` or the *class-key* `class`. A *standard-layout union* is a standard-layout class defined with the *class-key* `union`.

1

[*Note:* Standard-layout classes are useful for communicating with code written in other programming languages. Their layout is specified in 9.194. — *end note*]

1

108) Base class subobjects are not so constrained.

109) This ensures that two subobjects that have the same class type and that belong to the same most derived object are not allocated at the same address (5.3247).

A *POD struct*[110] is a non-union class that is both a trivial class and a standard-layout class, and has no non-static data members of type non-POD struct, non-POD union (or array of such types). Similarly, a *POD union* is a union that is both a trivial class and a standard-layout class, and has no non-static data members of type non-POD struct, non-POD union (or array of such types). A *POD class* is a class that is either a POD struct or a POD union.

[*Example:*

```
struct N {          // neither trivial nor standard-layout
  int i;
  int j;
  virtual ~N();
};

struct T {          // trivial but not standard-layout
  int i;
private:
  int j;
};

struct SL {         // standard-layout but not trivial
  int i;
  int j;
  ~SL();
};

struct POD {        // both trivial and standard-layout
  int i;
  int j;
};
```

— *end example*]

1

If a *class-head-name* contains a *nested-name-specifier*, the *class-specifier* shall refer to a class that was previously declared directly in the class or namespace to which the *nested-name-specifier* refers, or in an element of the inline namespace set (7.1149.1) of that namespace (i.e., not merely inherited or introduced by a *using-declaration*), and the *class-specifier* shall appear in a namespace enclosing the previous declaration. In such cases, the *nested-name-specifier* of the *class-head-name* of the definition shall not begin with a *decltype-specifier*.

9.172 Class names [class.name]

1 A class definition introduces a new type. [*Example:*

```
struct X { int a; };
struct Y { int a; };
X a1;
Y a2;
int a3;
```

declares three variables of three different types. This implies that

```
a1 = a2;                    // error: Y assigned to X
a1 = a3;                    // error: int assigned to X
```

110) The acronym POD stands for "plain old data".

are type mismatches, and that

```
int f(X);
int f(Y);
```

declare an overloaded (Clause 13) function `f()` and not simply a single function `f()` twice. For the same reason,

```
struct S { int a; };
struct S { int a; };            // error, double definition
```

is ill-formed because it defines `S` twice. — *end example*]

2

A class declaration introduces the class name into the scope where it is declared and hides any class, variable, function, or other declaration of that name in an enclosing scope (3.464). If a class name is declared in a scope where a variable, function, or enumerator of the same name is also declared, then when both declarations are in scope, the class can be referred to only using an *elaborated-type-specifier* (3.1395.1). [*Example:*

```
struct stat {
  // ...
};

stat gstat;                     // use plain stat to
                                // define variable

int stat(struct stat*);         // redeclare stat as function

void f() {
  struct stat* ps;              // struct prefix needed
                                // to name struct stat
  stat(ps);                     // call stat()
}
```

— *end example*] A *declaration* consisting solely of *class-key identifier;* is either a redeclaration of the name in the current scope or a forward declaration of the identifier as a class name. It introduces the class name into the current scope. [*Example:*

```
struct s { int a; };

void g() {
  struct s;                     // hide global struct s
                                // with a block-scope declaration
  s* p;                         // refer to local struct s
  struct s { char* p; };        // define local struct s
  struct s;                     // redeclaration, has no effect
}
```

— *end example*] [*Note:* Such declarations allow definition of classes that refer to each other. [*Example:*

```
class Vector;

class Matrix {
  // ...
  friend Vector operator*(const Matrix&, const Vector&);
};
```

```
class Vector {
  // ...
  friend Vector operator*(const Matrix&, const Vector&);
};
```

Declaration of `friend`s is described in 11.94, operator functions in 13.1122. — *end example*] — *end note*]

1

[*Note:* An *elaborated-type-specifier* (7.808.0.1) can also be used as a *type-specifier* as part of a declaration. It differs from a class declaration in that if a class of the elaborated name is in scope the elaborated name will refer to it. — *end note*] [*Example:*

```
struct s { int a; };

void g(int s) {
  struct s* p = new struct s;    // global s
  p->a = s;                      // parameter s
}
```

— *end example*]

1 [*Note:* The declaration of a class name takes effect immediately after the *identifier* is seen in the class definition or *elaborated-type-specifier*. For example,

```
class A * A;
```

first specifies `A` to be the name of a class and then redefines it as the name of a pointer to an object of that class. This means that the elaborated form `class A` must be used to refer to the class. Such artistry with names can be confusing and is best avoided. — *end note*]

2

A *typedef-name* (7.235.1) that names a class type, or a cv-qualified version thereof, is also a *class-name*. If a *typedef-name* that names a cv-qualified class type is used where a *class-name* is required, the cv-qualifiers are ignored. A *typedef-name* shall not be used as the *identifier* in a *class-head*.

9.194 Class members [class.mem]

member-specification:
- *member-declaration member-specification*$_{opt}$
- *access-specifier* : *member-specification*$_{opt}$

member-declaration:
- *attribute-specifier-seq*$_{opt}$ *decl-specifier-seq*$_{opt}$ *member-declarator-list*$_{opt}$;
- *function-definition*
- *using-declaration*
- *static_assert-declaration*
- *template-declaration*
- *alias-declaration*
- *empty-declaration*

member-declarator-list:
- *member-declarator*
- *member-declarator-list* , *member-declarator*

member-declarator:
- *declarator virt-specifier-seq*$_{opt}$ *pure-specifier*$_{opt}$
- *declarator brace-or-equal-initializer*$_{opt}$
- *identifier*$_{opt}$ *attribute-specifier-seq*$_{opt}$: *constant-expression*

virt-specifier-seq:
    virt-specifier
    virt-specifier-seq virt-specifier
virt-specifier:
    `override`
    `final`
pure-specifier:
    `= 0`

1

The *member-specification* in a class definition declares the full set of members of the class; no member can be added elsewhere. Members of a class are data members, member functions (9.401), nested types, and enumerators. Data members and member functions are static or non-static; see 9.559. Nested types are classes (9.172, 9.960) and enumerations (7.1001) defined in the class, and arbitrary types declared as members by use of a typedef declaration (7.235.1). The enumerators of an unscoped enumeration (7.1001) defined in the class are members of the class. Except when used to declare friends (11.94), to declare an unnamed bit-field (9.921), or to introduce the name of a member of a base class into a derived class (7.1311.1), or when the declaration is an *empty-declaration*, *member-declaration*s declare members of the class, and each such *member-declaration* shall declare at least one member name of the class. A member shall not be declared twice in the *member-specification*, except that a nested class or member class template can be declared and then later defined, and except that an enumeration can be introduced with an *opaque-enum-declaration* and later redeclared with an *enum-specifier*.

1

A class is considered a completely-defined object type (3.2955) (or complete type) at the closing `}` of the *class-specifier*. Within the class *member-specification*, the class is regarded as complete within function bodies, default arguments, *using-declaration*s introducing inheriting constructors (12.2400), *exception-specification*s, and *brace-or-equal-initializer*s for non-static data members (including such things in nested classes). Otherwise it is regarded as incomplete within its own class *member-specification*.

1 [*Note:* A single name can denote several function members provided their types are sufficiently different (Clause 13). — *end note*]

2

A *brace-or-equal-initializer* shall appear only in the declaration of a data member. (For static data members, see 9.595.1; for non-static data members, see 12.1026.1).

1 A member shall not be declared with the `extern` or `register` *storage-class-specifier*. Within a class definition, a member shall not be declared with the `thread_local` *storage-class-specifier* unless also declared `static`.

2 The *decl-specifier-seq* may be omitted in constructor, destructor, and conversion function declarations only; when declaring another kind of member the *decl-specifier-seq* shall contain a *type-specifier* that is not a *cv-qualifier*. The *member-declarator-list* can be omitted only after a *class-specifier* or an *enum-specifier* or in a `friend` declaration (11.94). A *pure-specifier* shall be used only in the declaration of a virtual function (10.113).

3 The optional *attribute-specifier-seq* in a *member-declaration* appertains to each of the entities declared by the *member-declarator*s; it shall not appear if the optional *member-declarator-list* is omitted.

4

A *virt-specifier-seq* shall contain at most one of each *virt-specifier*. A *virt-specifier-seq* shall appear only in the declaration of a virtual member function (10.113).

1

Non-`static` (9.559) data members shall not have incomplete types. In particular, a class `C` shall not contain a non-static member of class `C`, but it can contain a pointer or reference to an object of class `C`.

1 [*Note:* See 5.398 for restrictions on the use of non-static data members and non-static member functions. — *end note*]

2

[*Note:* The type of a non-static member function is an ordinary function type, and the type of a non-static data member is an ordinary object type. There are no special member function types or data member types. — *end note*]

1

[*Example:* A simple example of a class definition is

```
struct tnode {
  char tword[20];
  int count;
  tnode* left;
  tnode* right;
};
```

which contains an array of twenty characters, an integer, and two pointers to objects of the same type. Once this definition has been given, the declaration

```
tnode s, *sp;
```

declares `s` to be a `tnode` and `sp` to be a pointer to a `tnode`. With these declarations, `sp->count` refers to the `count` member of the object to which `sp` points; `s.left` refers to the `left` subtree pointer of the object `s`; and `s.right->tword[0]` refers to the initial character of the `tword` member of the `right` subtree of `s`. — *end example*]

1

Nonstatic data members of a (non-union) class with the same access control (Clause 11) are allocated so that later members have higher addresses within a class object. The order of allocation of non-static data members with different access control is unspecified (Clause 11). Implementation alignment requirements might cause two adjacent members not to be allocated immediately after each other; so might requirements for space for managing virtual functions (10.113) and virtual base classes (10.31).

1

If `T` is the name of a class, then each of the following shall have a name different from `T`:

— every static data member of class `T`;

— every member function of class `T` [*Note:* This restriction does not apply to constructors, which do not have names (12.30) — *end note*];

— every member of class `T` that is itself a type;

— every enumerator of every member of class `T` that is an unscoped enumerated type; and

— every member of every anonymous union that is a member of class `T`.

1

In addition, if class `T` has a user-declared constructor (12.30), every non-static data member of class `T` shall have a name different from `T`.

1

Two standard-layout struct (Clause 9) types are *layout-compatible* if they have the same number of non-static data members and corresponding non-static data members (in declaration order) have layout-compatible types (3.2955).

1 Two standard-layout union (Clause 9) types are *layout-compatible* if they have the same number of non-static data members and corresponding non-static data members (in any order) have layout-compatible types (3.2955).

2

If a standard-layout union contains two or more standard-layout structs that share a common initial sequence, and if the standard-layout union object currently contains one of these standard-layout structs, it is permitted to inspect the common initial part of any of them. Two standard-layout structs share a common initial sequence if corresponding members have layout-compatible types and either neither member is a bit-field or both are bit-fields with the same width for a sequence of one or more initial members.

1

If a standard-layout class object has any non-static data members, its address is the same as the address of its first non-static data member. Otherwise, its address is the same as the address of its first base class subobject (if any). [*Note:* There might therefore be unnamed padding within a standard-layout struct object, but not at its beginning, as necessary to achieve appropriate alignment. — *end note*]

9.401 Member functions [class.mfct]

1

Functions declared in the definition of a class, excluding those declared with a `friend` specifier (11.94), are called member functions of that class. A member function may be declared `static` in which case it is a *static* member function of its class (9.559); otherwise it is a *non-static* member function of its class (9.465.1, 9.508.1).

1

A member function may be defined (8.706) in its class definition, in which case it is an *inline* member function (7.94.1), or it may be defined outside of its class definition if it has already been declared but not defined in its class definition. A member function definition that appears outside of the class definition shall appear in a namespace scope enclosing the class definition. Except for member function definitions that appear outside of a class definition, and except for explicit specializations of member functions of class templates and member function templates (14.1614) appearing outside of the class definition, a member function shall not be redeclared.

1

An `inline` member function (whether static or non-static) may also be defined outside of its class definition provided either its declaration in the class definition or its definition outside of the class definition declares the function as `inline`. [*Note:* Member functions of a class in namespace scope have external linkage. Member functions of a local class (9.975) have no linkage. See 3.1459. — *end note*]

1

There shall be at most one definition of a non-inline member function in a program; no diagnostic is required. There may be more than one `inline` member function definition in a program. See 3.91 and 7.94.1.

1

If the definition of a member function is lexically outside its class definition, the member function name shall be qualified by its class name using the `::` operator. [*Note:* A name used in a member function definition (that is, in the *parameter-declaration-clause* including the default arguments (8.545.1) or in the member function body) is looked up as described in 3.923. — *end note*] [*Example:*

```
struct X {
  typedef int T;
  static T count;
  void f(T);
};
```

```
void X::f(T t = count) { }
```

The member function `f` of class `X` is defined in global scope; the notation `X::f` specifies that the function `f` is a member of class `X` and in the scope of class `X`. In the function definition, the parameter type `T` refers to the typedef member `T` declared in class `X` and the default argument `count` refers to the static data member `count` declared in class `X`. — *end example*]

1

A `static` local variable in a member function always refers to the same object, whether or not the member function is `inline`.

1

Previously declared member functions may be mentioned in `friend` declarations.

1

Member functions of a local class shall be defined inline in their class definition, if they are defined at all.

1

[*Note:* A member function can be declared (but not defined) using a typedef for a function type. The resulting member function has exactly the same type as it would have if the function declarator were provided explicitly, see 8.360.1. For example,

```
typedef void fv(void);
typedef void fvc(void) const;
struct S {
  fv memfunc1;      // equivalent to: void memfunc1(void);
  void memfunc2();
  fvc memfunc3;     // equivalent to: void memfunc3(void) const;
};
fv  S::* pmfv1 = &S::memfunc1;
fv  S::* pmfv2 = &S::memfunc2;
fvc S::* pmfv3 = &S::memfunc3;
```

Also see 14.326. — *end note*]

9.465.1 Nonstatic member functions [class.mfct.non-static]

1

A *non-static* member function may be called for an object of its class type, or for an object of a class derived (Clause 10) from its class type, using the class member access syntax (5.1194.1, 13.230.0.1). A non-static member function may also be called directly using the function call syntax (5.955.1, 13.230.0.1) from within the body of a member function of its class or of a class derived from its class.

1

If a non-static member function of a class `X` is called for an object that is not of type `X`, or of a type derived from `X`, the behavior is undefined.

1

When an *id-expression* (5.398) that is not part of a class member access syntax (5.1194.1) and not used to form a pointer to member (5.1898.1) is used in a member of class `X` in a context where `this` can be used (5.398.1), if name lookup (3.923) resolves the name in the *id-expression* to a non-static non-type member of some class `C`, and if either the *id-expression* is potentially evaluated or `C` is `X` or a base class of `X`, the *id-expression* is transformed into a class member access expression (5.1194.1) using `(*this)` (9.508.1) as the *postfix-expression* to the left of the `.` operator. [*Note:* If `C` is not `X` or a base class of `X`, the class member access expression is ill-formed. — *end note*] Similarly during name lookup, when an *unqualified-id* (5.398)

used in the definition of a member function for class `X` resolves to a `static` member, an enumerator or a nested type of class `X` or of a base class of `X`, the *unqualified-id* is transformed into a *qualified-id* (5.398) in which the *nested-name-specifier* names the class of the member function. [*Example:*

```
struct tnode {
  char tword[20];
  int count;
  tnode* left;
  tnode* right;
  void set(const char*, tnode* l, tnode* r);
};

void tnode::set(const char* w, tnode* l, tnode* r) {
  count = strlen(w)+1;
  if (sizeof(tword)<=count)
      perror("tnode string too long");
  strcpy(tword,w);
  left = l;
  right = r;
}

void f(tnode n1, tnode n2) {
  n1.set("abc",&n2,0);
  n2.set("def",0,0);
}
```

In the body of the member function `tnode::set`, the member names `tword`, `count`, `left`, and `right` refer to members of the object for which the function is called. Thus, in the call `n1.set("abc",&n2,0)`, `tword` refers to `n1.tword`, and in the call `n2.set("def",0,0)`, it refers to `n2.tword`. The functions `strlen`, `perror`, and `strcpy` are not members of the class `tnode` and should be declared elsewhere.[111] — *end example*]

1

A non-static member function may be declared `const`, `volatile`, or `const volatile`. These *cv-qualifiers* affect the type of the `this` pointer (9.508.1). They also affect the function type (8.360.1) of the member function; a member function declared `const` is a *const* member function, a member function declared `volatile` is a *volatile* member function and a member function declared `const volatile` is a *const volatile* member function. [*Example:*

```
struct X {
  void g() const;
  void h() const volatile;
};
```

`X::g` is a `const` member function and `X::h` is a `const volatile` member function. — *end example*]

1

A non-static member function may be declared with a *ref-qualifier* (8.360.1); see 13.133.1.

1 A non-static member function may be declared *virtual* (10.113) or *pure virtual* (10.276).

9.508.1 The `this` pointer [class.this]

1

In the body of a non-static (9.401) member function, the keyword `this` is a prvalue expression whose value is the address of the object for which the function is called. The type of `this` in a member function of

111) See, for example, `<cstring>` (21.261).

a class `X` is `X*`. If the member function is declared `const`, the type of `this` is `const X*`, if the member function is declared `volatile`, the type of `this` is `volatile X*`, and if the member function is declared `const volatile`, the type of `this` is `const volatile X*`. [*Note:* thus in a `const` member function, the object for which the function is called is accessed through a `const` access path. — *end note*] [*Example:*

```
struct s {
  int a;
  int f() const;
  int g() { return a++; }
  int h() const { return a++; } // error
};

int s::f() const { return a; }
```

The `a++` in the body of `s::h` is ill-formed because it tries to modify (a part of) the object for which `s::h()` is called. This is not allowed in a `const` member function because `this` is a pointer to `const`; that is, `*this` has `const` type. — *end example*]

1

Similarly, `volatile` semantics (7.567.0.1) apply in `volatile` member functions when accessing the object and its non-static data members.

1

A *cv-qualified* member function can be called on an object-expression (5.1194.1) only if the object-expression is as cv-qualified or less-cv-qualified than the member function. [*Example:*

```
void k(s& x, const s& y) {
  x.f();
  x.g();
  y.f();
  y.g();                    // error
}
```

The call `y.g()` is ill-formed because `y` is `const` and `s::g()` is a non-`const` member function, that is, `s::g()` is less-qualified than the object-expression `y`. — *end example*]

1 Constructors (12.30) and destructors (12.807) shall not be declared `const`, `volatile` or `const volatile`. [*Note:* However, these functions can be invoked to create and destroy objects with cv-qualified types, see (12.30) and (12.807). — *end note*]

9.559 Static members [class.static]

1

A data or function member of a class may be declared `static` in a class definition, in which case it is a *static member* of the class.

1

A `static` member `s` of class `X` may be referred to using the *qualified-id* expression `X::s`; it is not necessary to use the class member access syntax (5.1194.1) to refer to a `static` member. A `static` member may be referred to using the class member access syntax, in which case the object expression is evaluated. [*Example:*

```
struct process {
  static void reschedule();
};
process& g();
```

```
void f() {
  process::reschedule();        // OK: no object necessary
  g().reschedule();             // g() is called
}
```

— *end example*]

1

A `static` member may be referred to directly in the scope of its class or in the scope of a class derived (Clause 10) from its class; in this case, the `static` member is referred to as if a *qualified-id* expression was used, with the *nested-name-specifier* of the *qualified-id* naming the class scope from which the static member is referenced. [*Example:*

```
int g();
struct X {
  static int g();
};
struct Y : X {
  static int i;
};
int Y::i = g();                 // equivalent to Y::g();
```

— *end example*]

1

If an *unqualified-id* (5.398) is used in the definition of a `static` member following the member's *declarator-id*, and name lookup (3.952.1) finds that the *unqualified-id* refers to a `static` member, enumerator, or nested type of the member's class (or of a base class of the member's class), the *unqualified-id* is transformed into a *qualified-id* expression in which the *nested-name-specifier* names the class scope from which the member is referenced. [*Note:* See 5.398 for restrictions on the use of non-static data members and non-static member functions. — *end note*]

1

Static members obey the usual class member access rules (Clause 11). When used in the declaration of a class member, the `static` specifier shall only be used in the member declarations that appear within the *member-specification* of the class definition. [*Note:* It cannot be specified in member declarations that appear in namespace scope. — *end note*]

9.583.1 Static member functions [class.static.mfct]

1 [*Note:* The rules described in 9.401 apply to `static` member functions. — *end note*]

2

[*Note:* A `static` member function does not have a `this` pointer (9.508.1). — *end note*] A `static` member function shall not be `virtual`. There shall not be a `static` and a non-static member function with the same name and the same parameter types (13.12). A `static` member function shall not be declared `const`, `volatile`, or `const volatile`.

9.595.1 Static data members [class.static.data]

1

A `static` data member is not part of the subobjects of a class. If a `static` data member is declared `thread_local` there is one copy of the member per thread. If a `static` data member is not declared `thread_local` there is one copy of the data member that is shared by all the objects of the class.

1

The declaration of a `static` data member in its class definition is not a definition and may be of an incomplete type other than cv-qualified `void`. The definition for a `static` data member shall appear in a namespace scope enclosing the member's class definition. In the definition at namespace scope, the name of the `static` data member shall be qualified by its class name using the `::` operator. The *initializer* expression in the definition of a `static` data member is in the scope of its class (3.760.1). [*Example:*

```
class process {
  static process* run_chain;
  static process* running;
};

process* process::running = get_main();
process* process::run_chain = running;
```

The `static` data member `run_chain` of class `process` is defined in global scope; the notation `process::run_chain` specifies that the member `run_chain` is a member of class `process` and in the scope of class `process`. In the `static` data member definition, the *initializer* expression refers to the `static` data member `running` of class `process`. — *end example*]

[*Note:* Once the `static` data member has been defined, it exists even if no objects of its class have been created. [*Example:* in the example above, `run_chain` and `running` exist even if no objects of class `process` are created by the program. — *end example*] — *end note*]

1

If a non-volatile `const static` data member is of integral or enumeration type, its declaration in the class definition can specify a *brace-or-equal-initializer* in which every *initializer-clause* that is an *assignment-expression* is a constant expression (5.3643). A `static` data member of literal type can be declared in the class definition with the `constexpr` specifier; if so, its declaration shall specify a *brace-or-equal-initializer* in which every *initializer-clause* that is an *assignment-expression* is a constant expression. [*Note:* In both these cases, the member may appear in constant expressions. — *end note*] The member shall still be defined in a namespace scope if it is odr-used (3.91) in the program and the namespace scope definition shall not contain an *initializer*.

1

[*Note:* There shall be exactly one definition of a `static` data member that is odr-used (3.91) in a program; no diagnostic is required. — *end note*] Unnamed classes and classes contained directly or indirectly within unnamed classes shall not contain `static` data members.

1

`Static` data members of a class in namespace scope have external linkage (3.1459). A local class shall not have `static` data members.

1

`Static` data members are initialized and destroyed exactly like non-local variables (3.1937.1, 3.2200.1).

1 A `static` data member shall not be `mutable` (7.49.1).

9.815 Unions [class.union]

1

In a union, at most one of the non-static data members can be active at any time, that is, the value of at most one of the non-static data members can be stored in a union at any time. [*Note:* One special guarantee is made in order to simplify the use of unions: If a standard-layout union contains several standard-layout structs that share a common initial sequence (9.194), and if an object of this standard-layout union type contains one of the standard-layout structs, it is permitted to inspect the common initial sequence of any

of standard-layout struct members; see 9.194. — *end note*] The size of a union is sufficient to contain the largest of its non-static data members. Each non-static data member is allocated as if it were the sole member of a struct. All non-static data members of a union object have the same address.

1

A union can have member functions (including constructors and destructors), but not virtual (10.113) functions. A union shall not have base classes. A union shall not be used as a base class. If a union contains a non-static data member of reference type the program is ill-formed. [*Note:* If any non-static data member of a union has a non-trivial default constructor (12.30), copy constructor (12.1577), move constructor (12.1577), copy assignment operator (12.1577), move assignment operator (12.1577), or destructor (12.807), the corresponding member function of the union must be user-provided or it will be implicitly deleted (8.793.1) for the union. — *end note*]

1

[*Example:* Consider the following union:

```
union U {
  int i;
  float f;
  std::string s;
};
```

Since `std::string` (21.26) declares non-trivial versions of all of the special member functions, `U` will have an implicitly deleted default constructor, copy/move constructor, copy/move assignment operator, and destructor. To use `U`, some or all of these member functions must be user-provided. — *end example*]

1

[*Note:* In general, one must use explicit destructor calls and placement new operators to change the active member of a union. — *end note*] [*Example:* Consider an object `u` of a `union` type `U` having non-static data members `m` of type `M` and `n` of type `N`. If `M` has a non-trivial destructor and `N` has a non-trivial constructor (for instance, if they declare or inherit virtual functions), the active member of `u` can be safely switched from `m` to `n` using the destructor and placement new operator as follows:

```
u.m.~M();
new (&u.n) N;
```

— *end example*]

1

A union of the form

`union {` *member-specification* `} ;`

is called an *anonymous union*; it defines an unnamed object of unnamed type. The *member-specification* of an anonymous union shall only define non-static data members. [*Note:* Nested types, anonymous unions, and functions cannot be declared within an anonymous union. — *end note*] The names of the members of an anonymous union shall be distinct from the names of any other entity in the scope in which the anonymous union is declared. For the purpose of name lookup, after the anonymous union definition, the members of the anonymous union are considered to have been defined in the scope in which the anonymous union is declared. [*Example:*

```
void f() {
  union { int a; const char* p; };
  a = 1;
  p = "Jennifer";
}
```

Here `a` and `p` are used like ordinary (nonmember) variables, but since they are union members they have the same address. — *end example*]

1

Anonymous unions declared in a named namespace or in the global namespace shall be declared `static`. Anonymous unions declared at block scope shall be declared with any storage class allowed for a block-scope variable, or with no storage class. A storage class is not allowed in a declaration of an anonymous union in a class scope. An anonymous union shall not have `private` or `protected` members (Clause 11). An anonymous union shall not have function members.

1 A union for which objects, pointers, or references are declared is not an anonymous union. [*Example:*

```
void f() {
  union { int aa; char* p; } obj, *ptr = &obj;
  aa = 1;                          // error
  ptr->aa = 1;                     // OK
}
```

The assignment to plain `aa` is ill-formed since the member name is not visible outside the union, and even if it were visible, it is not associated with any particular object. — *end example*] [*Note:* Initialization of unions with no user-declared constructors is described in (8.1817.1). — *end note*]

2

A *union-like class* is a union or a class that has an anonymous union as a direct member. A union-like class `X` has a set of *variant members*. If `X` is a union, a non-static data member of `X` that is not an anonymous union is a variant member of `X`. In addition, a non-static data member of an anonymous union that is a member of `X` is also a variant member of `X`. At most one variant member of a union may have a *brace-or-equal-initializer*. [*Example:*

```
union U {
  int x = 0;
  union { };
  union {
    int z;
    int y = 1; // error: initialization for second variant member of U
  };
};
```

— *end example*]

9.921 Bit-fields [class.bit]

1

A *member-declarator* of the form

$$\textit{identifier}_{opt}\ \textit{attribute-specifier-seq}_{opt}\texttt{:}\ \textit{constant-expression}$$

specifies a bit-field; its length is set off from the bit-field name by a colon. The optional *attribute-specifier-seq* appertains to the entity being declared. The bit-field attribute is not part of the type of the class member. The *constant-expression* shall be an integral constant expression with a value greater than or equal to zero. The value of the integral constant expression may be larger than the number of bits in the object representation (3.2955) of the bit-field's type; in such cases the extra bits are used as padding bits and do not participate in the value representation (3.2955) of the bit-field. Allocation of bit-fields within a class object is implementation-defined. Alignment of bit-fields is implementation-defined. Bit-fields are packed into some addressable allocation unit. [*Note:* Bit-fields straddle allocation units on some machines and not on others. Bit-fields are assigned right-to-left on some machines, left-to-right on others. — *end note*]

1

A declaration for a bit-field that omits the *identifier* declares an *unnamed* bit-field. Unnamed bit-fields are not members and cannot be initialized. [*Note:* An unnamed bit-field is useful for padding to conform to externally-imposed layouts. — *end note*] As a special case, an unnamed bit-field with a width of zero specifies alignment of the next bit-field at an allocation unit boundary. Only when declaring an unnamed bit-field may the value of the *constant-expression* be equal to zero.

1

A bit-field shall not be a static member. A bit-field shall have integral or enumeration type (3.3151.1). A `bool` value can successfully be stored in a bit-field of any nonzero size. The address-of operator `&` shall not be applied to a bit-field, so there are no pointers to bit-fields. A non-const reference shall not be bound to a bit-field (8.2156.1). [*Note:* If the initializer for a reference of type `const T&` is an lvalue that refers to a bit-field, the reference is bound to a temporary initialized to hold the value of the bit-field; the reference is not bound to the bit-field directly. See 8.2156.1. — *end note*]

1

If the value `true` or `false` is stored into a bit-field of type `bool` of any size (including a one bit bit-field), the original `bool` value and the value of the bit-field shall compare equal. If the value of an enumerator is stored into a bit-field of the same enumeration type and the number of bits in the bit-field is large enough to hold all the values of that enumeration type (7.1001), the original enumerator value and the value of the bit-field shall compare equal. [*Example:*

```
enum BOOL { FALSE=0, TRUE=1 };
struct A {
  BOOL b:1;
};
A a;
void f() {
  a.b = TRUE;
  if (a.b == TRUE)              // yields true
    { /* ... */ }
}
```

— *end example*]

9.960 Nested class declarations [class.nest]

1

A class can be declared within another class. A class declared within another is called a *nested* class. The name of a nested class is local to its enclosing class. The nested class is in the scope of its enclosing class. [*Note:* See 5.398 for restrictions on the use of non-static data members and non-static member functions. — *end note*]

[*Example:*

```
int x;
int y;

struct enclose {
  int x;
  static int s;

  struct inner {
    void f(int i) {
      int a = sizeof(x);        // OK: operand of sizeof is an unevaluated operand
```

```
      x = i;                    // error: assign to enclose::x
      s = i;                    // OK: assign to enclose::s
      ::x = i;                  // OK: assign to global x
      y = i;                    // OK: assign to global y
    }
    void g(enclose* p, int i) {
      p->x = i;                 // OK: assign to enclose::x
    }
  };
};

inner* p = 0;                   // error: inner not in scope
```

— *end example*]

1

Member functions and static data members of a nested class can be defined in a namespace scope enclosing the definition of their class. [*Example:*

```
struct enclose {
  struct inner {
    static int x;
    void f(int i);
  };
};

int enclose::inner::x = 1;

void enclose::inner::f(int i) { /* ... */ }
```

— *end example*]

1

If class `X` is defined in a namespace scope, a nested class `Y` may be declared in class `X` and later defined in the definition of class `X` or be later defined in a namespace scope enclosing the definition of class `X`. [*Example:*

```
class E {
  class I1;                     // forward declaration of nested class
  class I2;
  class I1 { };                 // definition of nested class
};
class E::I2 { };                // definition of nested class
```

— *end example*]

1

Like a member function, a friend function (11.94) defined within a nested class is in the lexical scope of that class; it obeys the same rules for name binding as a static member function of that class (9.559), but it has no special access rights to members of an enclosing class.

9.975 Local class declarations [class.local]

1

A class can be declared within a function definition; such a class is called a *local* class. The name of a local class is local to its enclosing scope. The local class is in the scope of the enclosing scope, and has the same

access to names outside the function as does the enclosing function. Declarations in a local class shall not odr-use (3.91) a variable with automatic storage duration from an enclosing scope. [*Example:*

```
int x;
void f() {
  static int s ;
  int x;
  const int N = 5;
  extern int q();

  struct local {
    int g() { return x; }        // error: odr-use of automatic variable x
    int h() { return s; }        // OK
    int k() { return ::x; }      // OK
    int l() { return q(); }      // OK
    int m() { return N; }        // OK: not an odr-use
    int* n() { return &N; }      // error: odr-use of automatic variable N
  };
}

local* p = 0;                    // error: local not in scope
```

— *end example*]

1

An enclosing function has no special access to members of the local class; it obeys the usual access rules (Clause 11). Member functions of a local class shall be defined within their class definition, if they are defined at all.

1

If class `X` is a local class a nested class `Y` may be declared in class `X` and later defined in the definition of class `X` or be later defined in the same scope as the definition of class `X`. A class nested within a local class is a local class.

1 A local class shall not have static data members.

9.992 Nested type names [class.nested.type]

1

Type names obey exactly the same scope rules as other names. In particular, type names defined within a class definition cannot be used outside their class without qualification. [*Example:*

```
struct X {
  typedef int I;
  class Y { /* ... */ };
  I a;
};

I b;                             // error
Y c;                             // error
X::Y d;                          // OK
X::I e;                          // OK
```

— *end example*]

10 Derived classes [class.derived]

1 A list of base classes can be specified in a class definition using the notation:

base-clause:
: *base-specifier-list*

base-specifier-list:
base-specifier ...$_{opt}$
base-specifier-list , *base-specifier* ...$_{opt}$

base-specifier:
attribute-specifier-seq$_{opt}$ *base-type-specifier*
attribute-specifier-seq$_{opt}$ `virtual` *access-specifier*$_{opt}$ *base-type-specifier*
attribute-specifier-seq$_{opt}$ *access-specifier* `virtual`$_{opt}$ *base-type-specifier*

class-or-decltype:
nested-name-specifier$_{opt}$ *class-name*
decltype-specifier

base-type-specifier:
class-or-decltype

access-specifier:
`private`
`protected`
`public`

The optional *attribute-specifier-seq* appertains to the *base-specifier*.

2

The type denoted by a *base-type-specifier* shall be a class type that is not an incompletely defined class (Clause 9); this class is called a *direct base class* for the class being defined. During the lookup for a base class name, non-type names are ignored (3.875.1). If the name found is not a *class-name*, the program is ill-formed. A class `B` is a base class of a class `D` if it is a direct base class of `D` or a direct base class of one of `D`'s base classes. A class is an *indirect* base class of another if it is a base class but not a direct base class. A class is said to be (directly or indirectly) *derived* from its (direct or indirect) base classes. [*Note:* See Clause 11 for the meaning of *access-specifier*. — *end note*] Unless redeclared in the derived class, members of a base class are also considered to be members of the derived class. The base class members are said to be *inherited* by the derived class. Inherited members can be referred to in expressions in the same manner as other members of the derived class, unless their names are hidden or ambiguous (10.52). [*Note:* The scope resolution operator `::` (5.398) can be used to refer to a direct or indirect base member explicitly. This allows access to a name that has been redeclared in the derived class. A derived class can itself serve as a base class subject to access control; see 11.43. A pointer to a derived class can be implicitly converted to a pointer to an accessible unambiguous base class (4.579). An lvalue of a derived class type can be bound to a reference to an accessible unambiguous base class (8.2156.1). — *end note*]

1

The *base-specifier-list* specifies the type of the *base class subobjects* contained in an object of the derived class type. [*Example:*

```
struct Base {
  int a, b, c;
};

struct Derived : Base {
```

```
  int b;
};

struct Derived2 : Derived {
  int c;
};
```

Here, an object of class `Derived2` will have a subobject of class `Derived` which in turn will have a subobject of class `Base`. — *end example*]

1 A *base-specifier* followed by an ellipsis is a pack expansion (14.703.1).

2

The order in which the base class subobjects are allocated in the most derived object (1.319) is unspecified. [*Note:* a derived class and its base class subobjects can be represented by a directed acyclic graph (DAG) where an arrow means "directly derived from." A DAG of subobjects is often referred to as a "subobject lattice."

Figure 2 — Directed acyclic graph

1 The arrows need not have a physical representation in memory. — *end note*]

2 [*Note:* Initialization of objects representing base classes can be specified in constructors; see 12.1026.1. — *end note*]

3

[*Note:* A base class subobject might have a layout (3.2235) different from the layout of a most derived object of the same type. A base class subobject might have a polymorphic behavior (12.1483) different from the polymorphic behavior of a most derived object of the same type. A base class subobject may be of zero size (Clause 9); however, two subobjects that have the same class type and that belong to the same most derived object must not be allocated at the same address (5.3247). — *end note*]

10.31 Multiple base classes [class.mi]

1

A class can be derived from any number of base classes. [*Note:* The use of more than one direct base class is often called multiple inheritance. — *end note*] [*Example:*

```
class A { /* ... */ };
class B { /* ... */ };
class C { /* ... */ };
class D : public A, public B, public C { /* ... */ };
```

— *end example*]

1

[*Note:* The order of derivation is not significant except as specified by the semantics of initialization by constructor (12.1026.1), cleanup (12.807), and storage layout (9.194, 11.36). — *end note*]

1

A class shall not be specified as a direct base class of a derived class more than once. [*Note:* A class can be an indirect base class more than once and can be a direct and an indirect base class. There are limited things that can be done with such a class. The non-static data members and member functions of the direct base class cannot be referred to in the scope of the derived class. However, the static members, enumerations and types can be unambiguously referred to. — *end note*] [*Example:*

```
class X { /* ... */ };
class Y : public X, public X { /* ... */ };              // ill-formed

class L { public: int next;  /* ... */ };
class A : public L { /* ... */ };
class B : public L { /* ... */ };
class C : public A, public B { void f(); /* ... */ };    // well-formed
class D : public A, public L { void f(); /* ... */ };    // well-formed
```

— *end example*]

1

A base class specifier that does not contain the keyword `virtual`, specifies a *non-virtual* base class. A base class specifier that contains the keyword `virtual`, specifies a *virtual* base class. For each distinct occurrence of a non-virtual base class in the class lattice of the most derived class, the most derived object (1.319) shall contain a corresponding distinct base class subobject of that type. For each distinct base class that is specified virtual, the most derived object shall contain a single base class subobject of that type. [*Example:* for an object of class type `C`, each distinct occurrence of a (non-virtual) base class `L` in the class lattice of `C` corresponds one-to-one with a distinct `L` subobject within the object of type `C`. Given the class `C` defined above, an object of class `C` will have two subobjects of class `L` as shown below.

Figure 3 — Non-virtual base

1

In such lattices, explicit qualification can be used to specify which subobject is meant. The body of function `C::f` could refer to the member `next` of each `L` subobject:

```
void C::f() { A::next = B::next; }        // well-formed
```

Without the `A::` or `B::` qualifiers, the definition of `C::f` above would be ill-formed because of ambiguity (10.52).

1

For another example,

```
class V { /* ... */ };
class A : virtual public V { /* ... */ };
class B : virtual public V { /* ... */ };
class C : public A, public B { /* ... */ };
```

for an object `c` of class type `C`, a single subobject of type `V` is shared by every base subobject of `c` that has a `virtual` base class of type `V`. Given the class `C` defined above, an object of class `C` will have one subobject of class `V`, as shown below.

Figure 4 — Virtual base

1 A class can have both virtual and non-virtual base classes of a given type.

```
class B { /* ... */ };
class X : virtual public B { /* ... */ };
class Y : virtual public B { /* ... */ };
class Z : public B { /* ... */ };
class AA : public X, public Y, public Z { /* ... */ };
```

For an object of class `AA`, all `virtual` occurrences of base class `B` in the class lattice of `AA` correspond to a single `B` subobject within the object of type `AA`, and every other occurrence of a (non-virtual) base class `B` in the class lattice of `AA` corresponds one-to-one with a distinct `B` subobject within the object of type `AA`. Given the class `AA` defined above, class `AA` has two subobjects of class `B`: `Z`'s `B` and the virtual `B` shared by `X` and `Y`, as shown below.

Figure 5 — Virtual and non-virtual base

— *end example*]

10.52 Member name lookup [class.member.lookup]

1

Member name lookup determines the meaning of a name (*id-expression*) in a class scope (3.760.1). Name lookup can result in an *ambiguity*, in which case the program is ill-formed. For an *id-expression*, name

lookup begins in the class scope of `this`; for a *qualified-id*, name lookup begins in the scope of the *nested-name-specifier*. Name lookup takes place before access control (3.923, Clause 11).

1

The following steps define the result of name lookup for a member name `f` in a class scope `C`.

1

The *lookup set* for `f` in `C`, called $S(f, C)$, consists of two component sets: the *declaration set*, a set of members named `f`; and the *subobject set*, a set of subobjects where declarations of these members (possibly including *using-declarations*) were found. In the declaration set, *using-declarations* are replaced by the set of designated members that are not hidden or overridden by members of the derived class (7.1311.1), and type declarations (including injected-class-names) are replaced by the types they designate. $S(f, C)$ is calculated as follows:

1

If `C` contains a declaration of the name `f`, the declaration set contains every declaration of `f` declared in `C` that satisfies the requirements of the language construct in which the lookup occurs. [*Note:* Looking up a name in an *elaborated-type-specifier* (3.1395.1) or *base-specifier* (Clause 10), for instance, ignores all non-type declarations, while looking up a name in a *nested-name-specifier* (3.1236.1) ignores function, variable, and enumerator declarations. As another example, looking up a name in a *using-declaration* (7.1311.1) includes the declaration of a class or enumeration that would ordinarily be hidden by another declaration of that name in the same scope. — *end note*] If the resulting declaration set is not empty, the subobject set contains `C` itself, and calculation is complete.

1

Otherwise (i.e., `C` does not contain a declaration of `f` or the resulting declaration set is empty), $S(f, C)$ is initially empty. If `C` has base classes, calculate the lookup set for `f` in each direct base class subobject B_i, and merge each such lookup set $S(f, B_i)$ in turn into $S(f, C)$.

1

The following steps define the result of merging lookup set $S(f, B_i)$ into the intermediate $S(f, C)$:

— If each of the subobject members of $S(f, B_i)$ is a base class subobject of at least one of the subobject members of $S(f, C)$, or if $S(f, B_i)$ is empty, $S(f, C)$ is unchanged and the merge is complete. Conversely, if each of the subobject members of $S(f, C)$ is a base class subobject of at least one of the subobject members of $S(f, B_i)$, or if $S(f, C)$ is empty, the new $S(f, C)$ is a copy of $S(f, B_i)$.

— Otherwise, if the declaration sets of $S(f, B_i)$ and $S(f, C)$ differ, the merge is ambiguous: the new $S(f, C)$ is a lookup set with an invalid declaration set and the union of the subobject sets. In subsequent merges, an invalid declaration set is considered different from any other.

— Otherwise, the new $S(f, C)$ is a lookup set with the shared set of declarations and the union of the subobject sets.

1

The result of name lookup for `f` in `C` is the declaration set of $S(f, C)$. If it is an invalid set, the program is ill-formed. [*Example:*

```
struct A { int x; };                    // S(x,A) = { { A::x }, { A } }
struct B { float x; };                  // S(x,B) = { { B::x }, { B } }
struct C: public A, public B { };       // S(x,C) = { invalid, { A in C, B in C } }
struct D: public virtual C { };         // S(x,D) = S(x,C)
struct E: public virtual C { char x; }; // S(x,E) = { { E::x }, { E } }
struct F: public D, public E { };       // S(x,F) = S(x,E)
int main() {
```

```
  F f;
  f.x = 0;                          // OK, lookup finds E::x
}
```

$S(x, F)$ is unambiguous because the `A` and `B` base subobjects of `D` are also base subobjects of `E`, so $S(x, D)$ is discarded in the first merge step. — *end example*]

1

If the name of an overloaded function is unambiguously found, overloading resolution (13.88) also takes place before access control. Ambiguities can often be resolved by qualifying a name with its class name. [*Example:*

```
struct A {
  int f();
};

struct B {
  int f();
};

struct C : A, B {
  int f() { return A::f() + B::f(); }
};
```

— *end example*]

1

[*Note:* A static member, a nested type or an enumerator defined in a base class `T` can unambiguously be found even if an object has more than one base class subobject of type `T`. Two base class subobjects share the non-static member subobjects of their common virtual base classes. — *end note*] [*Example:*

```
struct V {
  int v;
};
struct A {
  int a;
  static int   s;
  enum { e };
};
struct B : A, virtual V { };
struct C : A, virtual V { };
struct D : B, C { };

void f(D* pd) {
  pd->v++;          // OK: only one v (virtual)
  pd->s++;          // OK: only one s (static)
  int i = pd->e;    // OK: only one e (enumerator)
  pd->a++;          // error, ambiguous: two as in D
}
```

— *end example*]

1 [*Note:* When virtual base classes are used, a hidden declaration can be reached along a path through the subobject lattice that does not pass through the hiding declaration. This is not an ambiguity. The identical use with non-virtual base classes is an ambiguity; in that case there is no unique instance of the name that hides all the others. — *end note*] [*Example:*

```
struct V { int f();  int x; };
struct W { int g();  int y; };
struct B : virtual V, W {
  int f();  int x;
  int g();  int y;
};
struct C : virtual V, W { };

struct D : B, C { void glorp(); };
```

Figure 6 — Name lookup

2 [*Note:* The names declared in `V` and the left-hand instance of `W` are hidden by those in `B`, but the names declared in the right-hand instance of `W` are not hidden at all. — *end note*]

```
void D::glorp() {
  x++;              // OK: B::x hides V::x
  f();              // OK: B::f() hides V::f()
  y++;              // error: B::y and C's W::y
  g();              // error: B::g() and C's W::g()
}
```

— *end example*]

3 An explicit or implicit conversion from a pointer to or an expression designating an object of a derived class to a pointer or reference to one of its base classes shall unambiguously refer to a unique object representing the base class. [*Example:*

```
struct V { };
struct A { };
struct B : A, virtual V { };
struct C : A, virtual V { };
struct D : B, C { };

void g() {
  D d;
  B* pb = &d;
  A* pa = &d;       // error, ambiguous: C's A or B's A?
  V* pv = &d;       // OK: only one V subobject
}
```

— *end example*]

4

[*Note:* Even if the result of name lookup is unambiguous, use of a name found in multiple subobjects might still be ambiguous (4.710, 5.1194.1, 11.43). — *end note*] [*Example:*

```
struct B1 {
  void f();
  static void f(int);
  int i;
};
struct B2 {
  void f(double);
};
struct I1: B1 { };
struct I2: B1 { };

struct D: I1, I2, B2 {
  using B1::f;
  using B2::f;
  void g() {
    f();                        // Ambiguous conversion of this
    f(0);                       // Unambiguous (static)
    f(0.0);                     // Unambiguous (only one B2)
    int B1::* mpB1 = &D::i;     // Unambiguous
    int D::* mpD = &D::i;       // Ambiguous conversion
  }
};
```

— *end example*]

10.113 Virtual functions [class.virtual]

1

Virtual functions support dynamic binding and object-oriented programming. A class that declares or inherits a virtual function is called a *polymorphic class*.

1

If a virtual member function `vf` is declared in a class `Base` and in a class `Derived`, derived directly or indirectly from `Base`, a member function `vf` with the same name, parameter-type-list (8.360.1), cv-qualification, and ref-qualifier (or absence of same) as `Base::vf` is declared, then `Derived::vf` is also virtual (whether or not it is so declared) and it *overrides*[112] `Base::vf`. For convenience we say that any virtual function overrides itself. A virtual member function `C::vf` of a class object `S` is a *final overrider* unless the most derived class (1.319) of which `S` is a base class subobject (if any) declares or inherits another member function that overrides `vf`. In a derived class, if a virtual member function of a base class subobject has more than one final overrider the program is ill-formed. [*Example:*

```
struct A {
  virtual void f();
};
struct B : virtual A {
  virtual void f();
};
struct C : B , virtual A {
  using A::f;
};

void foo() {
```

112) A function with the same name but a different parameter list (Clause 13) as a virtual function is not necessarily virtual and does not override. The use of the `virtual` specifier in the declaration of an overriding function is legal but redundant (has empty semantics). Access control (Clause 11) is not considered in determining overriding.

```
  C c;
  c.f();              // calls B::f, the final overrider
  c.C::f();           // calls A::f because of the using-declaration
}
```

— *end example*]

[*Example:*

```
struct A { virtual void f(); };
struct B : A { };
struct C : A { void f(); };
struct D : B, C { };  // OK: A::f and C::f are the final overriders
                      // for the B and C subobjects, respectively
```

— *end example*]

1

[*Note:* A virtual member function does not have to be visible to be overridden, for example,

```
struct B {
  virtual void f();
};
struct D : B {
  void f(int);
};
struct D2 : D {
  void f();
};
```

the function `f(int)` in class `D` hides the virtual function `f()` in its base class `B`; `D::f(int)` is not a virtual function. However, `f()` declared in class `D2` has the same name and the same parameter list as `B::f()`, and therefore is a virtual function that overrides the function `B::f()` even though `B::f()` is not visible in class `D2`. — *end note*]

1

If a virtual function `f` in some class `B` is marked with the *virt-specifier* `final` and in a class `D` derived from `B` a function `D::f` overrides `B::f`, the program is ill-formed. [*Example:*

```
struct B {
  virtual void f() const final;
};

struct D : B {
  void f() const;     // error: D::f attempts to override final B::f
};
```

— *end example*]

1

If a virtual function is marked with the *virt-specifier* `override` and does not override a member function of a base class, the program is ill-formed. [*Example:*

```
struct B {
  virtual void f(int);
};
```

```
struct D : B {
  virtual void f(long) override;  // error: wrong signature overriding B::f
  virtual void f(int) override;   // OK
};
```

— *end example*]

1

Even though destructors are not inherited, a destructor in a derived class overrides a base class destructor declared virtual; see 12.807 and 12.988.

1

The return type of an overriding function shall be either identical to the return type of the overridden function or *covariant* with the classes of the functions. If a function `D::f` overrides a function `B::f`, the return types of the functions are covariant if they satisfy the following criteria:

— both are pointers to classes, both are lvalue references to classes, or both are rvalue references to classes[113]

— the class in the return type of `B::f` is the same class as the class in the return type of `D::f`, or is an unambiguous and accessible direct or indirect base class of the class in the return type of `D::f`

— both pointers or references have the same cv-qualification and the class type in the return type of `D::f` has the same cv-qualification as or less cv-qualification than the class type in the return type of `B::f`.

1

If the class type in the covariant return type of `D::f` differs from that of `B::f`, the class type in the return type of `D::f` shall be complete at the point of declaration of `D::f` or shall be the class type `D`. When the overriding function is called as the final overrider of the overridden function, its result is converted to the type returned by the (statically chosen) overridden function (5.955.1). [*Example:*

```
class B { };
class D : private B { friend class Derived; };
struct Base {
  virtual void vf1();
  virtual void vf2();
  virtual void vf3();
  virtual B*   vf4();
  virtual B*   vf5();
  void f();
};

struct No_good : public Base {
  D*  vf4();        // error: B (base class of D) inaccessible
};

class A;
struct Derived : public Base {
    void vf1();     // virtual and overrides Base::vf1()
    void vf2(int);  // not virtual, hides Base::vf2()
    char vf3();     // error: invalid difference in return type only
    D*  vf4();      // OK: returns pointer to derived class
    A*  vf5();      // error: returns pointer to incomplete class
```

113) Multi-level pointers to classes or references to multi-level pointers to classes are not allowed.

```
    void f();
};

void g() {
  Derived d;
  Base* bp = &d;                // standard conversion:
                                // Derived* to Base*
  bp->vf1();                    // calls Derived::vf1()
  bp->vf2();                    // calls Base::vf2()
  bp->f();                      // calls Base::f() (not virtual)
  B*  p = bp->vf4();            // calls Derived::pf() and converts the
                                // result to B*
  Derived*  dp = &d;
  D*  q = dp->vf4();            // calls Derived::pf() and does not
                                // convert the result to B*
  dp->vf2();                    // ill-formed: argument mismatch
}
```

— *end example*]

1

[*Note:* The interpretation of the call of a virtual function depends on the type of the object for which it is called (the dynamic type), whereas the interpretation of a call of a non-virtual member function depends only on the type of the pointer or reference denoting that object (the static type) (5.955.1). — *end note*]

1

[*Note:* The `virtual` specifier implies membership, so a virtual function cannot be a nonmember (7.94.1) function. Nor can a virtual function be a static member, since a virtual function call relies on a specific object for determining which function to invoke. A virtual function declared in one class can be declared a `friend` in another class. — *end note*]

1

A virtual function declared in a class shall be defined, or declared pure (10.276) in that class, or both; but no diagnostic is required (3.91).

1 [*Example:* here are some uses of virtual functions with multiple base classes:

```
struct A {
  virtual void f();
};

struct B1 : A {                 // note non-virtual derivation
  void f();
};

struct B2 : A {
  void f();
};

struct D : B1, B2 {             // D has two separate A subobjects
};

void foo() {
  D   d;
// A* ap = &d; // would be ill-formed: ambiguous
  B1*  b1p = &d;
```

```
  A*   ap = b1p;
  D*   dp = &d;
  ap->f();                        // calls D::B1::f
  dp->f();                        // ill-formed: ambiguous
}
```

In class `D` above there are two occurrences of class `A` and hence two occurrences of the virtual member function `A::f`. The final overrider of `B1::A::f` is `B1::f` and the final overrider of `B2::A::f` is `B2::f`.

2

The following example shows a function that does not have a unique final overrider:

```
struct A {
  virtual void f();
};

struct VB1 : virtual A {          // note virtual derivation
  void f();
};

struct VB2 : virtual A {
  void f();
};

struct Error : VB1, VB2 {         // ill-formed
};

struct Okay : VB1, VB2 {
  void f();
};
```

Both `VB1::f` and `VB2::f` override `A::f` but there is no overrider of both of them in class `Error`. This example is therefore ill-formed. Class `Okay` is well formed, however, because `Okay::f` is a final overrider.

1 The following example uses the well-formed classes from above.

```
struct VB1a : virtual A {         // does not declare f
};

struct Da : VB1a, VB2 {
};

void foe() {
  VB1a*  vb1ap = new Da;
  vb1ap->f();                     // calls VB2::f
}
```

— *end example*]

2

Explicit qualification with the scope operator (5.398) suppresses the virtual call mechanism. [*Example:*

```
class B { public: virtual void f(); };
class D : public B { public: void f(); };

void D::f() { /* ... */ B::f(); }
```

Here, the function call in `D::f` really does call `B::f` and not `D::f`. — *end example*]

1

A function with a deleted definition (8.706) shall not override a function that does not have a deleted definition. Likewise, a function that does not have a deleted definition shall not override a function with a deleted definition.

10.276 Abstract classes [class.abstract]

1

The abstract class mechanism supports the notion of a general concept, such as a `shape`, of which only more concrete variants, such as `circle` and `square`, can actually be used. An abstract class can also be used to define an interface for which derived classes provide a variety of implementations.

1

An *abstract class* is a class that can be used only as a base class of some other class; no objects of an abstract class can be created except as subobjects of a class derived from it. A class is abstract if it has at least one *pure virtual function*. [*Note:* Such a function might be inherited: see below. — *end note*] A virtual function is specified *pure* by using a *pure-specifier* (9.194) in the function declaration in the class definition. A pure virtual function need be defined only if called with, or as if with (12.807), the *qualified-id* syntax (5.398). [*Example:*

```
class point { /* ... */ };
class shape {                    // abstract class
  point center;
public:
  point where() { return center; }
  void move(point p) { center=p; draw(); }
  virtual void rotate(int) = 0; // pure virtual
  virtual void draw() = 0;      // pure virtual
};
```

— *end example*] [*Note:* A function declaration cannot provide both a *pure-specifier* and a definition — *end note*] [*Example:*

```
struct C {
  virtual void f() = 0 { };     // ill-formed
};
```

— *end example*]

1

An abstract class shall not be used as a parameter type, as a function return type, or as the type of an explicit conversion. Pointers and references to an abstract class can be declared. [*Example:*

```
shape x;                        // error: object of abstract class
shape* p;                       // OK
shape f();                      // error
void g(shape);                  // error
shape& h(shape&);               // OK
```

— *end example*]

1

A class is abstract if it contains or inherits at least one pure virtual function for which the final overrider is pure virtual. [*Example:*

```
class ab_circle : public shape {
  int radius;
public:
  void rotate(int) { }
  // ab_circle::draw() is a pure virtual
};
```

Since `shape::draw()` is a pure virtual function `ab_circle::draw()` is a pure virtual by default. The alternative declaration,

```
class circle : public shape {
  int radius;
public:
  void rotate(int) { }
  void draw();                    // a definition is required somewhere
};
```

would make class `circle` nonabstract and a definition of `circle::draw()` must be provided. — *end example*]

1

[*Note:* An abstract class can be derived from a class that is not abstract, and a pure virtual function may override a virtual function which is not pure. — *end note*]

1

Member functions can be called from a constructor (or destructor) of an abstract class; the effect of making a virtual call (10.113) to a pure virtual function directly or indirectly for the object being created (or destroyed) from such a constructor (or destructor) is undefined.

11 Member access control [class.access]

1

A member of a class can be

— `private`; that is, its name can be used only by members and friends of the class in which it is declared.

— `protected`; that is, its name can be used only by members and friends of the class in which it is declared, by classes derived from that class, and by their friends (see 11.169).

— `public`; that is, its name can be used anywhere without access restriction.

1

A member of a class can also access all the names to which the class has access. A local class of a member function may access the same names that the member function itself may access.[114]

1

Members of a class defined with the keyword `class` are `private` by default. Members of a class defined with the keywords `struct` or `union` are `public` by default. [*Example:*

```
class X {
  int a;            // X::a is private by default
};

struct S {
  int a;            // S::a is public by default
};
```

— *end example*]

1

Access control is applied uniformly to all names, whether the names are referred to from declarations or expressions. [*Note:* Access control applies to names nominated by `friend` declarations (11.94) and *using-declarations* (7.1311.1). — *end note*] In the case of overloaded function names, access control is applied to the function selected by overload resolution. [*Note:* Because access control applies to names, if access control is applied to a typedef name, only the accessibility of the typedef name itself is considered. The accessibility of the entity referred to by the typedef is not considered. For example,

```
class A {
  class B { };
public:
  typedef B BB;
};

void f() {
  A::BB x;          // OK, typedef name A::BB is public
  A::B y;           // access error, A::B is private
}
```

114) Access permissions are thus transitive and cumulative to nested and local classes.

— *end note*]

1

It should be noted that it is *access* to members and base classes that is controlled, not their *visibility*. Names of members are still visible, and implicit conversions to base classes are still considered, when those members and base classes are inaccessible. The interpretation of a given construct is established without regard to access control. If the interpretation established makes use of inaccessible member names or base classes, the construct is ill-formed.

1

All access controls in Clause 11 affect the ability to access a class member name from the declaration of a particular entity, including parts of the declaration preceding the name of the entity being declared and, if the entity is a class, the definitions of members of the class appearing outside the class's *member-specification*. [*Note:* this access also applies to implicit references to constructors, conversion functions, and destructors. — *end note*] [*Example:*

```
class A {
  typedef int I;     // private member
  I f();
  friend I g(I);
  static I x;
  template<int> struct Q;
  template<int> friend struct R;
protected:
    struct B { };
};

A::I A::f() { return 0; }
A::I g(A::I p = A::x);
A::I g(A::I p) { return 0; }
A::I A::x = 0;
template<A::I> struct A::Q { };
template<A::I> struct R { };

struct D: A::B, A { };
```

1

Here, all the uses of `A::I` are well-formed because `A::f`, `A::x`, and `A::Q` are members of class `A` and `g` and `R` are friends of class `A`. This implies, for example, that access checking on the first use of `A::I` must be deferred until it is determined that this use of `A::I` is as the return type of a member of class `A`. Similarly, the use of `A::B` as a *base-specifier* is well-formed because `D` is derived from `A`, so checking of *base-specifier*s must be deferred until the entire *base-specifier-list* has been seen. — *end example*]

1

The names in a default argument (8.545.1) are bound at the point of declaration, and access is checked at that point rather than at any points of use of the default argument. Access checking for default arguments in function templates and in member functions of class templates is performed as described in 14.1644.1.

1 The names in a default *template-argument* (14.97) have their access checked in the context in which they appear rather than at any points of use of the default *template-argument*. [*Example:*

```
class B { };
template <class T> class C {
protected:
  typedef T TT;
```

```
};

template <class U, class V = typename U::TT>
class D : public U { };

D <C<B> >* d;        // access error, C::TT is protected
```

— *end example*]

11.36 Access specifiers [class.access.spec]

1 Member declarations can be labeled by an *access-specifier* (Clause 10):

access-specifier : *member-specification*$_{opt}$

An *access-specifier* specifies the access rules for members following it until the end of the class or until another *access-specifier* is encountered. [*Example:*

```
class X {
  int a;             // X::a is private by default: class used
public:
  int b;             // X::b is public
  int c;             // X::c is public
};
```

— *end example*]

2 Any number of access specifiers is allowed and no particular order is required. [*Example:*

```
struct S {
  int a;             // S::a is public by default: struct used
protected:
  int b;             // S::b is protected
private:
  int c;             // S::c is private
public:
  int d;             // S::d is public
};
```

— *end example*]

3 [*Note:* The effect of access control on the order of allocation of data members is described in 9.194. — *end note*]

4 When a member is redeclared within its class definition, the access specified at its redeclaration shall be the same as at its initial declaration. [*Example:*

```
struct S {
  class A;
  enum E : int;
private:
  class A { };         // error: cannot change access
  enum E: int { e0 }; // error: cannot change access
};
```

— *end example*]

5

[*Note:* In a derived class, the lookup of a base class name will find the injected-class-name instead of the name of the base class in the scope in which it was declared. The injected-class-name might be less accessible than the name of the base class in the scope in which it was declared. — *end note*]

[*Example:*

```
class A { };
class B : private A { };
class C : public B {
  A* p;              // error: injected-class-name A is inaccessible
  ::A* q;            // OK
};
```

— *end example*]

11.43 Accessibility of base classes and base class members [class.access.base]

1

If a class is declared to be a base class (Clause 10) for another class using the `public` access specifier, the `public` members of the base class are accessible as `public` members of the derived class and `protected` members of the base class are accessible as `protected` members of the derived class. If a class is declared to be a base class for another class using the `protected` access specifier, the `public` and `protected` members of the base class are accessible as `protected` members of the derived class. If a class is declared to be a base class for another class using the `private` access specifier, the `public` and `protected` members of the base class are accessible as `private` members of the derived class[115].

1

In the absence of an *access-specifier* for a base class, `public` is assumed when the derived class is defined with the *class-key* `struct` and `private` is assumed when the class is defined with the *class-key* `class`. [*Example:*

```
class B { /* ... */ };
class D1 : private B { /* ... */ };
class D2 : public B { /* ... */ };
class D3 : B { /* ... */ };      // B private by default
struct D4 : public B { /* ... */ };
struct D5 : private B { /* ... */ };
struct D6 : B { /* ... */ };     // B public by default
class D7 : protected B { /* ... */ };
struct D8 : protected B { /* ... */ };
```

Here `B` is a public base of `D2`, `D4`, and `D6`, a private base of `D1`, `D3`, and `D5`, and a protected base of `D7` and `D8`. — *end example*]

1

[*Note:* A member of a private base class might be inaccessible as an inherited member name, but accessible directly. Because of the rules on pointer conversions (4.579) and explicit casts (5.2765), a conversion from a pointer to a derived class to a pointer to an inaccessible base class might be ill-formed if an implicit conversion is used, but well-formed if an explicit cast is used. For example,

```
class B {
public:
  int mi;                    // non-static member
  static int si;             // static member
```

115) As specified previously in Clause 11, private members of a base class remain inaccessible even to derived classes unless `friend` declarations within the base class definition are used to grant access explicitly.

```
};
class D : private B {
};
class DD : public D {
  void f();
};

void DD::f() {
  mi = 3;                       // error: mi is private in D
  si = 3;                       // error: si is private in D
  ::B  b;
  b.mi = 3;                     // OK (b.mi is different from this->mi)
  b.si = 3;                     // OK (b.si is different from this->si)
  ::B::si = 3;                  // OK
  ::B* bp1 = this;              // error: B is a private base class
  ::B* bp2 = (::B*)this;        // OK with cast
  bp2->mi = 3;                  // OK: access through a pointer to B.
}
```

— *end note*]

1

A base class `B` of `N` is *accessible* at R, if

— an invented public member of `B` would be a public member of `N`, or

— R occurs in a member or friend of class `N`, and an invented public member of `B` would be a private or protected member of `N`, or

— R occurs in a member or friend of a class `P` derived from `N`, and an invented public member of `B` would be a private or protected member of `P`, or

— there exists a class `S` such that `B` is a base class of `S` accessible at R and `S` is a base class of `N` accessible at R.

[*Example:*

```
class B {
public:
  int m;
};

class S: private B {
  friend class N;
};

class N: private S {
  void f() {
    B* p = this;     // OK because class S satisfies the fourth condition
                     // above: B is a base class of N accessible in f() because
                     // B is an accessible base class of S and S is an accessible
                     // base class of N.
  }
};
```

— *end example*]

1

If a base class is accessible, one can implicitly convert a pointer to a derived class to a pointer to that base class (4.579, 4.710). [*Note:* It follows that members and friends of a class `X` can implicitly convert an `X*` to a pointer to a private or protected immediate base class of `X`. — *end note*] The access to a member is affected by the class in which the member is named. This naming class is the class in which the member name was looked up and found. [*Note:* This class can be explicit, e.g., when a *qualified-id* is used, or implicit, e.g., when a class member access operator (5.1194.1) is used (including cases where an implicit "`this->`" is added). If both a class member access operator and a *qualified-id* are used to name the member (as in `p->T::m`), the class naming the member is the class denoted by the *nested-name-specifier* of the *qualified-id* (that is, `T`). — *end note*] A member `m` is accessible at the point R when named in class `N` if

— `m` as a member of `N` is public, or

— `m` as a member of `N` is private, and R occurs in a member or friend of class `N`, or

— `m` as a member of `N` is protected, and R occurs in a member or friend of class `N`, or in a member or friend of a class `P` derived from `N`, where `m` as a member of `P` is public, private, or protected, or

— there exists a base class `B` of `N` that is accessible at R, and `m` is accessible at R when named in class `B`. [*Example:*

```
class B;
class A {
private:
  int i;
  friend void f(B*);
};
class B : public A { };
void f(B* p) {
  p->i = 1;          // OK: B* can be implicitly converted to A*,
                     // and f has access to i in A
}
```

— *end example*]

1

If a class member access operator, including an implicit "`this->`," is used to access a non-static data member or non-static member function, the reference is ill-formed if the left operand (considered as a pointer in the "`.`" operator case) cannot be implicitly converted to a pointer to the naming class of the right operand. [*Note:* This requirement is in addition to the requirement that the member be accessible as named. — *end note*]

11.94 Friends [class.friend]

1

A friend of a class is a function or class that is given permission to use the private and protected member names from the class. A class specifies its friends, if any, by way of friend declarations. Such declarations give special access rights to the friends, but they do not make the nominated friends members of the befriending class. [*Example:* the following example illustrates the differences between members and friends:

```
class X {
  int a;
  friend void friend_set(X*, int);
public:
```

```
  void member_set(int);
};

void friend_set(X* p, int i) { p->a = i; }
void X::member_set(int i) { a = i; }

void f() {
  X obj;
  friend_set(&obj,10);
  obj.member_set(10);
}
```

— *end example*]

1

Declaring a class to be a friend implies that the names of private and protected members from the class granting friendship can be accessed in the *base-specifier*s and member declarations of the befriended class. [*Example:*

```
class A {
  class B { };
  friend class X;
};

struct X : A::B {    // OK: A::B accessible to friend
  A::B mx;           // OK: A::B accessible to member of friend
  class Y {
    A::B my;         // OK: A::B accessible to nested member of friend
  };
};
```

— *end example*] [*Example:*

```
class X {
  enum { a=100 };
  friend class Y;
};

class Y {
  int v[X::a];       // OK, Y is a friend of X
};

class Z {
  int v[X::a];       // error: X::a is private
};
```

— *end example*]

A class shall not be defined in a friend declaration. [*Example:*

```
class A {
  friend class B { }; // error: cannot define class in friend declaration
};
```

— *end example*]

1

A `friend` declaration that does not declare a function shall have one of the following forms:

`friend` *elaborated-type-specifier* `;`
`friend` *simple-type-specifier* `;`
`friend` *typename-specifier* `;`

[*Note:* A `friend` declaration may be the *declaration* in a *template-declaration* (Clause 14, 14.755.1). — *end note*] If the type specifier in a `friend` declaration designates a (possibly cv-qualified) class type, that class is declared as a friend; otherwise, the `friend` declaration is ignored. [*Example:*

```
class C;
typedef C Ct;

class X1 {
  friend C;          // OK: class C is a friend
};

class X2 {
  friend Ct;         // OK: class C is a friend
  friend D;          // error: no type-name D in scope
  friend class D;    // OK: elaborated-type-specifier declares new class
};

template <typename T> class R {
  friend T;
};

R<C> rc;             // class C is a friend of R<C>
R<int> Ri;           // OK: "friend int;" is ignored
```

— *end example*]

1

A function first declared in a friend declaration has external linkage (3.1459). Otherwise, the function retains its previous linkage (7.49.1).

1

When a `friend` declaration refers to an overloaded name or operator, only the function specified by the parameter types becomes a friend. A member function of a class `X` can be a friend of a class `Y`. [*Example:*

```
class Y {
  friend char* X::foo(int);
  friend X::X(char);             // constructors can be friends
  friend X::~X();                // destructors can be friends
};
```

— *end example*]

1

A function can be defined in a friend declaration of a class if and only if the class is a non-local class (9.975), the function name is unqualified, and the function has namespace scope. [*Example:*

```
class M {
  friend void f() { }            // definition of global f, a friend of M,
                                 // not the definition of a member function
};
```

— *end example*]

1

Such a function is implicitly `inline`. A `friend` function defined in a class is in the (lexical) scope of the class in which it is defined. A friend function defined outside the class is not (3.952.1).

1 No *storage-class-specifier* shall appear in the *decl-specifier-seq* of a friend declaration.

2

A name nominated by a friend declaration shall be accessible in the scope of the class containing the friend declaration. The meaning of the friend declaration is the same whether the friend declaration appears in the `private`, `protected` or `public` (9.194) portion of the class *member-specification*.

1

Friendship is neither inherited nor transitive. [*Example:*

```
class A {
  friend class B;
  int a;
};

class B {
  friend class C;
};

class C  {
  void f(A* p) {
    p->a++;          // error: C is not a friend of A
                     // despite being a friend of a friend
  }
};

class D : public B  {
  void f(A* p) {
    p->a++;          // error: D is not a friend of A
                     // despite being derived from a friend
  }
};
```

— *end example*]

1

If a friend declaration appears in a local class (9.975) and the name specified is an unqualified name, a prior declaration is looked up without considering scopes that are outside the innermost enclosing non-class scope. For a friend function declaration, if there is no prior declaration, the program is ill-formed. For a friend class declaration, if there is no prior declaration, the class that is specified belongs to the innermost enclosing non-class scope, but if it is subsequently referenced, its name is not found by name lookup until a matching declaration is provided in the innermost enclosing non-class scope. [*Example:*

```
class X;
void a();
void f() {
  class Y;
  extern void b();
  class A {
  friend class X;   // OK, but X is a local class, not ::X
  friend class Y;   // OK
  friend class Z;   // OK, introduces local class Z
```

```
  friend void a();   // error, ::a is not considered
  friend void b();   // OK
  friend void c();   // error
  };
  X* px;             // OK, but ::X is found
  Z* pz;             // error, no Z is found
}
```

— *end example*]

11.169 Protected member access [class.protected]

1

An additional access check beyond those described earlier in Clause 11 is applied when a non-static data member or non-static member function is a protected member of its naming class (11.43)[116] As described earlier, access to a protected member is granted because the reference occurs in a friend or member of some class `C`. If the access is to form a pointer to member (5.1898.1), the *nested-name-specifier* shall denote `C` or a class derived from `C`. All other accesses involve a (possibly implicit) object expression (5.1194.1). In this case, the class of the object expression shall be `C` or a class derived from `C`. [*Example:*

```
class B {
protected:
  int i;
  static int j;
};

class D1 : public B {
};

class D2 : public B {
  friend void fr(B*,D1*,D2*);
  void mem(B*,D1*);
};

void fr(B* pb, D1* p1, D2* p2) {
  pb->i = 1;                    // ill-formed
  p1->i = 2;                    // ill-formed
  p2->i = 3;                    // OK (access through a D2)
  p2->B::i = 4;                 // OK (access through a D2, even though
                                // naming class is B)
  int B::* pmi_B = &B::i;       // ill-formed
  int B::* pmi_B2 = &D2::i;     // OK (type of &D2::i is int B::*)
  B::j = 5;                     // OK (because refers to static member)
  D2::j = 6;                    // OK (because refers to static member)
}

void D2::mem(B* pb, D1* p1) {
  pb->i = 1;                    // ill-formed
  p1->i = 2;                    // ill-formed
  i = 3;                        // OK (access through this)
  B::i = 4;                     // OK (access through this, qualification ignored)
  int B::* pmi_B = &B::i;       // ill-formed
  int B::* pmi_B2 = &D2::i;     // OK
  j = 5;                        // OK (because j refers to static member)
```

116) This additional check does not apply to other members, e.g., static data members or enumerator member constants.

```
  B::j = 6;                    // OK (because B::j refers to static member)
}

void g(B* pb, D1* p1, D2* p2) {
  pb->i = 1;                   // ill-formed
  p1->i = 2;                   // ill-formed
  p2->i = 3;                   // ill-formed
}
```

— *end example*]

11.203 Access to virtual functions [class.access.virt]

1

The access rules (Clause 11) for a virtual function are determined by its declaration and are not affected by the rules for a function that later overrides it. [*Example:*

```
class B {
public:
  virtual int f();
};

class D : public B {
private:
  int f();
};

void f() {
  D d;
  B* pb = &d;
  D* pd = &d;

  pb->f();                     // OK: B::f() is public,
                               // D::f() is invoked
  pd->f();                     // error: D::f() is private
}
```

— *end example*]

1

Access is checked at the call point using the type of the expression used to denote the object for which the member function is called (`B*` in the example above). The access of the member function in the class in which it was defined (`D` in the example above) is in general not known.

11.223 Multiple access [class.paths]

1

If a name can be reached by several paths through a multiple inheritance graph, the access is that of the path that gives most access. [*Example:*

```
class W { public: void f(); };
class A : private virtual W { };
class B : public virtual W { };
class C : public A, public B {
  void f() { W::f(); }         // OK
};
```

1

Since `W::f()` is available to `C::f()` along the public path through `B`, access is allowed. — *end example*]

11.226 Nested classes [class.access.nest]

1

A nested class is a member and as such has the same access rights as any other member. The members of an enclosing class have no special access to members of a nested class; the usual access rules (Clause 11) shall be obeyed. [*Example:*

```
class E {
  int x;
  class B { };

  class I {
    B b;                        // OK: E::I can access E::B
    int y;
    void f(E* p, int i) {
      p->x = i;                 // OK: E::I can access E::x
    }
  };

  int g(I* p) {
    return p->y;                // error: I::y is private
  }
};
```

— *end example*]

12 Special member functions [special]

1

The default constructor (12.30), copy constructor and copy assignment operator (12.1577), move constructor and move assignment operator (12.1577), and destructor (12.807) are *special member functions*. [*Note:* The implementation will implicitly declare these member functions for some class types when the program does not explicitly declare them. The implementation will implicitly define them if they are odr-used (3.91). See 12.30, 12.807 and 12.1577. — *end note*] An implicitly-declared special member function is declared at the closing `}` of the *class-specifier*. Programs shall not define implicitly-declared special member functions.

1 Programs may explicitly refer to implicitly-declared special member functions. [*Example:* a program may explicitly call, take the address of or form a pointer to member to an implicitly-declared special member function.

```
struct A { };                    // implicitly declared A::operator=
struct B : A {
  B& operator=(const B &);
};
B& B::operator=(const B& s) {
  this->A::operator=(s);         // well formed
  return *this;
}
```

— *end example*]

2 [*Note:* The special member functions affect the way objects of class type are created, copied, moved, and destroyed, and how values can be converted to values of other types. Often such special member functions are called implicitly. — *end note*]

3

Special member functions obey the usual access rules (Clause 11). [*Example:* declaring a constructor `protected` ensures that only derived classes and friends can create objects using it. — *end example*]

1

For a class, its non-static data members, its non-virtual direct base classes, and, if the class is not abstract (10.276), its virtual base classes are called its *potentially constructed subobjects*.

12.30 Constructors [class.ctor]

1

Constructors do not have names. A declaration of a constructor uses a function declarator (8.360.1) of the form

ptr-declarator (*parameter-declaration-clause*) *exception-specification*$_{opt}$ *attribute-specifier-seq*$_{opt}$

where the *ptr-declarator* consists solely of an *id-expression*, an optional *attribute-specifier-seq*, and optional surrounding parentheses, and the *id-expression* has one of the following forms:

— in a *member-declaration* that belongs to the *member-specification* of a class but is not a friend declaration (11.94), the *id-expression* is the injected-class-name (Clause 9) of the immediately-enclosing class;

— in a *member-declaration* that belongs to the *member-specification* of a class template but is not a friend declaration, the *id-expression* is a *class-name* that names the current instantiation (14.1319.0.1) of the immediately-enclosing class template; or

— in a declaration at namespace scope or in a friend declaration, the *id-expression* is a *qualified-id* that names a constructor (3.1282.0.1).

The *class-name* shall not be a *typedef-name*. In a constructor declaration, each *decl-specifier* in the optional *decl-specifier-seq* shall be `friend`, `inline`, `explicit`, or `constexpr`. [*Example:*

```
struct S {
  S();                // declares the constructor
};

S::S() { }            // defines the constructor
```

— *end example*]

1

A constructor is used to initialize objects of its class type. Because constructors do not have names, they are never found during name lookup; however an explicit type conversion using the functional notation (5.1108.1) will cause a constructor to be called to initialize an object. [*Note:* For initialization of objects of class type see 12.1011. — *end note*]

1

A constructor can be invoked for a `const`, `volatile` or `const volatile` object. `const` and `volatile` semantics (7.567.0.1) are not applied on an object under construction. They come into effect when the constructor for the most derived object (1.319) ends.

1

A *default* constructor for a class `X` is a constructor of class `X` that can be called without an argument. If there is no user-declared constructor for class `X`, a constructor having no parameters is implicitly declared as defaulted (8.706). An implicitly-declared default constructor is an `inline public` member of its class. A defaulted default constructor for class `X` is defined as deleted if:

— `X` is a union-like class that has a variant member with a non-trivial default constructor,

— any non-static data member with no *brace-or-equal-initializer* is of reference type,

— any non-variant non-static data member of const-qualified type (or array thereof) with no *brace-or-equal-initializer* does not have a user-provided default constructor,

— `X` is a union and all of its variant members are of const-qualified type (or array thereof),

— `X` is a non-union class and all members of any anonymous union member are of const-qualified type (or array thereof),

— any potentially constructed subobject, except for a non-static data member with a *brace-or-equal-initializer*, has class type `M` (or array thereof) and either `M` has no default constructor or overload resolution (13.88) as applied to `M`'s default constructor results in an ambiguity or in a function that is deleted or inaccessible from the defaulted default constructor, or

— any potentially constructed subobject has a type with a destructor that is deleted or inaccessible from the defaulted default constructor.

A default constructor is trivial if it is not user-provided and if:

— its class has no virtual functions (10.113) and no virtual base classes (10.31), and

— no non-static data member of its class has a *brace-or-equal-initializer*, and

— all the direct base classes of its class have trivial default constructors, and

— for all the non-static data members of its class that are of class type (or array thereof), each such class has a trivial default constructor.

Otherwise, the default constructor is *non-trivial*.

1

A default constructor that is defaulted and not defined as deleted is *implicitly defined* when it is odr-used (3.91) to create an object of its class type (1.319) or when it is explicitly defaulted after its first declaration. The implicitly-defined default constructor performs the set of initializations of the class that would be performed by a user-written default constructor for that class with no *ctor-initializer* (12.1026.1) and an empty *compound-statement*. If that user-written default constructor would be ill-formed, the program is ill-formed. If that user-written default constructor would satisfy the requirements of a `constexpr` constructor (7.328.1), the implicitly-defined default constructor is `constexpr`. Before the defaulted default constructor for a class is implicitly defined, all the non-user-provided default constructors for its base classes and its non-static data members shall have been implicitly defined. [*Note:* An implicitly-declared default constructor has an *exception-specification* (15.228). An explicitly-defaulted definition might have an implicit *exception-specification,* see 8.706. — *end note*]

1

Default constructors are called implicitly to create class objects of static, thread, or automatic storage duration (3.2242.1, 3.2291.1, 3.2301.1) defined without an initializer (8.821), are called to create class objects of dynamic storage duration (3.2331.1) created by a *new-expression* in which the *new-initializer* is omitted (5.2221.1), or are called when the explicit type conversion syntax (5.1108.1) is used. A program is ill-formed if the default constructor for an object is implicitly used and the constructor is not accessible (Clause 11).

1 [*Note:* 12.1026.1 describes the order in which constructors for base classes and non-static data members are called and describes how arguments can be specified for the calls to these constructors. — *end note*]

2

A `return` statement in the body of a constructor shall not specify a return value. The address of a constructor shall not be taken.

1

A functional notation type conversion (5.1108.1) can be used to create new objects of its type. [*Note:* The syntax looks like an explicit call of the constructor. — *end note*] [*Example:*

```
complex zz = complex(1,2.3);
cprint( complex(7.8,1.2) );
```

— *end example*]

1 An object created in this way is unnamed. [*Note:* 12.306 describes the lifetime of temporary objects. — *end note*] [*Note:* Explicit constructor calls do not yield lvalues, see 3.3673. — *end note*]

2 [*Note:* some language constructs have special semantics when used during construction; see 12.1026.1 and 12.1483. — *end note*]

3

During the construction of a `const` object, if the value of the object or any of its subobjects is accessed through a glvalue that is not obtained, directly or indirectly, from the constructor's `this` pointer, the value of the object or subobject thus obtained is unspecified. [*Example:*

```
struct C;
void no_opt(C*);

struct C {
  int c;
  C() : c(0) { no_opt(this); }
};

const C cobj;

void no_opt(C* cptr) {
  int i = cobj.c * 100;          // value of cobj.c is unspecified
  cptr->c = 1;
  cout << cobj.c * 100           // value of cobj.c is unspecified
       << '\n';
}
```

— *end example*]

12.306 Temporary objects [class.temporary]

1

Temporaries of class type are created in various contexts: binding a reference to a prvalue (8.2156.1), returning a prvalue (6.270.2), a conversion that creates a prvalue (4.127, 5.1443.1, 5.1871.1, 5.2765), throwing an exception (15.7), and in some initializations (8.821). [*Note:* The lifetime of exception objects is described in 15.7. — *end note*] Even when the creation of the temporary object is unevaluated (Clause 5) or otherwise avoided (12.1577), all the semantic restrictions shall be respected as if the temporary object had been created and later destroyed. [*Note:* This includes accessibility (11) and whether it is deleted, for the constructor selected and for the destructor. However, in the special case of a function call used as the operand of a *decltype-specifier* (5.955.1), no temporary is introduced, so the foregoing does not apply to the prvalue of any such function call. — *end note*]

1

[*Example:* Consider the following code:

```
class X {
public:
  X(int);
  X(const X&);
  X& operator=(const X&);
  ~X();
};

class Y {
public:
  Y(int);
  Y(Y&&);
  ~Y();
};

X f(X);
Y g(Y);
```

```
void h() {
  X a(1);
  X b = f(X(2));
  Y c = g(Y(3));
  a = f(a);
}
```

An implementation might use a temporary in which to construct `X(2)` before passing it to `f()` using `X`'s copy constructor; alternatively, `X(2)` might be constructed in the space used to hold the argument. Likewise, an implementation might use a temporary in which to construct `Y(3)` before passing it to `g()` using `Y`'s move constructor; alternatively, `Y(3)` might be constructed in the space used to hold the argument. Also, a temporary might be used to hold the result of `f(X(2))` before copying it to `b` using `X`'s copy constructor; alternatively, `f()`'s result might be constructed in `b`. Likewise, a temporary might be used to hold the result of `g(Y(3))` before moving it to `c` using `Y`'s move constructor; alternatively, `g()`'s result might be constructed in `c`. On the other hand, the expression `a=f(a)` requires a temporary for the result of `f(a)`, which is then assigned to `a`. — *end example*]

1

When an implementation introduces a temporary object of a class that has a non-trivial constructor (12.30, 12.1577), it shall ensure that a constructor is called for the temporary object. Similarly, the destructor shall be called for a temporary with a non-trivial destructor (12.807). Temporary objects are destroyed as the last step in evaluating the full-expression (1.430) that (lexically) contains the point where they were created. This is true even if that evaluation ends in throwing an exception. The value computations and side effects of destroying a temporary object are associated only with the full-expression, not with any specific subexpression.

1

There are two contexts in which temporaries are destroyed at a different point than the end of the full-expression. The first context is when a default constructor is called to initialize an element of an array. If the constructor has one or more default arguments, the destruction of every temporary created in a default argument is sequenced before the construction of the next array element, if any.

1

The second context is when a reference is bound to a temporary.[117] The temporary to which the reference is bound or the temporary that is the complete object of a subobject to which the reference is bound persists for the lifetime of the reference except:

— A temporary bound to a reference member in a constructor's *ctor-initializer* (12.1026.1) persists until the constructor exits.

— A temporary bound to a reference parameter in a function call (5.955.1) persists until the completion of the full-expression containing the call.

— The lifetime of a temporary bound to the returned value in a function return statement (6.270.2) is not extended; the temporary is destroyed at the end of the full-expression in the return statement.

— A temporary bound to a reference in a *new-initializer* (5.2221.1) persists until the completion of the full-expression containing the *new-initializer*. [*Example:*

```
struct S { int mi; const std::pair<int,int>& mp; };
S a { 1, {2,3} };
S* p = new S{ 1, {2,3} };    // Creates dangling reference
```

117) The same rules apply to initialization of an `initializer_list` object (8.2435.1) with its underlying temporary array

— *end example*] [*Note:* This may introduce a dangling reference, and implementations are encouraged to issue a warning in such a case. — *end note*]

The destruction of a temporary whose lifetime is not extended by being bound to a reference is sequenced before the destruction of every temporary which is constructed earlier in the same full-expression. If the lifetime of two or more temporaries to which references are bound ends at the same point, these temporaries are destroyed at that point in the reverse order of the completion of their construction. In addition, the destruction of temporaries bound to references shall take into account the ordering of destruction of objects with static, thread, or automatic storage duration (3.2242.1, 3.2291.1, 3.2301.1); that is, if `obj1` is an object with the same storage duration as the temporary and created before the temporary is created the temporary shall be destroyed before `obj1` is destroyed; if `obj2` is an object with the same storage duration as the temporary and created after the temporary is created the temporary shall be destroyed after `obj2` is destroyed. [*Example:*

```
struct S {
  S();
  S(int);
  friend S operator+(const S&, const S&);
  ~S();
};
S obj1;
const S& cr = S(16)+S(23);
S obj2;
```

the expression `S(16) + S(23)` creates three temporaries: a first temporary `T1` to hold the result of the expression `S(16)`, a second temporary `T2` to hold the result of the expression `S(23)`, and a third temporary `T3` to hold the result of the addition of these two expressions. The temporary `T3` is then bound to the reference `cr`. It is unspecified whether `T1` or `T2` is created first. On an implementation where `T1` is created before `T2`, `T2` shall be destroyed before `T1`. The temporaries `T1` and `T2` are bound to the reference parameters of `operator+`; these temporaries are destroyed at the end of the full-expression containing the call to `operator+`. The temporary `T3` bound to the reference `cr` is destroyed at the end of `cr`'s lifetime, that is, at the end of the program. In addition, the order in which `T3` is destroyed takes into account the destruction order of other objects with static storage duration. That is, because `obj1` is constructed before `T3`, and `T3` is constructed before `obj2`, `obj2` shall be destroyed before `T3`, and `T3` shall be destroyed before `obj1`. — *end example*]

12.659 Conversions [class.conv]

1

Type conversions of class objects can be specified by constructors and by conversion functions. These conversions are called *user-defined conversions* and are used for implicit type conversions (Clause 4), for initialization (8.821), and for explicit type conversions (5.2765, 5.1443.1).

1

User-defined conversions are applied only where they are unambiguous (10.52, 12.742.1). Conversions obey the access control rules (Clause 11). Access control is applied after ambiguity resolution (3.923).

1

[*Note:* See 13.88 for a discussion of the use of conversions in function calls as well as examples below. — *end note*]

1

At most one user-defined conversion (constructor or conversion function) is implicitly applied to a single value.

[*Example:*

```
struct X {
  operator int();
};

struct Y {
  operator X();
};

Y a;
int b = a;          // error
                    // a.operator X().operator int() not tried
int c = X(a);       // OK: a.operator X().operator int()
```

— *end example*]

1

User-defined conversions are used implicitly only if they are unambiguous. A conversion function in a derived class does not hide a conversion function in a base class unless the two functions convert to the same type. Function overload resolution (13.562.1) selects the best conversion function to perform the conversion. [*Example:*

```
struct X {
  operator int();
};

struct Y : X {
    operator char();
};

void f(Y& a) {
  if (a) {          // ill-formed:
                    // X::operator int() or Y::operator char()
  }
}
```

— *end example*]

12.708.1 Conversion by constructor [class.conv.ctor]

1

A constructor declared without the *function-specifier* `explicit` specifies a conversion from the types of its parameters to the type of its class. Such a constructor is called a *converting constructor*. [*Example:*

```
struct X {
    X(int);
    X(const char*, int =0);
    X(int, int);
};

void f(X arg) {
  X a = 1;          // a = X(1)
  X b = "Jessie";   // b = X("Jessie",0)
  a = 2;            // a = X(2)
  f(3);             // f(X(3))
  f({1, 2});        // f(X(1,2))
}
```

— *end example*]

1

An explicit constructor constructs objects just like non-explicit constructors, but does so only where the direct-initialization syntax (8.821) or where casts (5.1443.1, 5.2765) are explicitly used. A default constructor may be an explicit constructor; such a constructor will be used to perform default-initialization or value-initialization (8.821). [*Example:*

```
struct Z {
  explicit Z();
  explicit Z(int);
  explicit Z(int, int);
};

Z a;                          // OK: default-initialization performed
Z a1 = 1;                     // error: no implicit conversion
Z a3 = Z(1);                  // OK: direct initialization syntax used
Z a2(1);                      // OK: direct initialization syntax used
Z* p = new Z(1);              // OK: direct initialization syntax used
Z a4 = (Z)1;                  // OK: explicit cast used
Z a5 = static_cast<Z>(1);     // OK: explicit cast used
Z a6 = { 3, 4 };              // error: no implicit conversion
```

— *end example*]

1

A non-explicit copy/move constructor (12.1577) is a converting constructor. An implicitly-declared copy/-move constructor is not an explicit constructor; it may be called for implicit type conversions.

12.742.1 Conversion functions [class.conv.fct]

1

A member function of a class `X` having no parameters with a name of the form

conversion-function-id:
`operator` *conversion-type-id*

conversion-type-id:
type-specifier-seq conversion-declarator$_{opt}$

conversion-declarator:
ptr-operator conversion-declarator$_{opt}$

specifies a conversion from `X` to the type specified by the *conversion-type-id*. Such functions are called conversion functions. No return type can be specified. If a conversion function is a member function, the type of the conversion function (8.360.1) is "function taking no parameter returning *conversion-type-id*". A conversion function is never used to convert a (possibly cv-qualified) object to the (possibly cv-qualified) same object type (or a reference to it), to a (possibly cv-qualified) base class of that type (or a reference to it), or to (possibly cv-qualified) void.[118]

[*Example:*

```
struct X {
  operator int();
};
```

118) These conversions are considered as standard conversions for the purposes of overload resolution (13.779.0.1, 13.870.0.0.1) and therefore initialization (8.821) and explicit casts (5.1443.1). A conversion to `void` does not invoke any conversion function (5.1443.1). Even though never directly called to perform a conversion, such conversion functions can be declared and can potentially be reached through a call to a virtual conversion function in a base class.

```
void f(X a) {
  int i = int(a);
  i = (int)a;
  i = a;
}
```

In all three cases the value assigned will be converted by `X::operator int()`. — *end example*]

1

A conversion function may be explicit (7.94.1), in which case it is only considered as a user-defined conversion for direct-initialization (8.821). Otherwise, user-defined conversions are not restricted to use in assignments and initializations. [*Example:*

```
class Y { };
struct Z {
  explicit operator Y() const;
};

void h(Z z) {
  Y y1(z);          // OK: direct-initialization
  Y y2 = z;         // ill-formed: copy-initialization
  Y y3 = (Y)z;      // OK: cast notation
}

void g(X a, X b) {
  int i = (a) ? 1+a : 0;
  int j = (a&&b) ? a+b : i;
  if (a) {
  }
}
```

— *end example*]

1

The *conversion-type-id* shall not represent a function type nor an array type. The *conversion-type-id* in a *conversion-function-id* is the longest possible sequence of *conversion-declarator*s. [*Note:* This prevents ambiguities between the declarator operator * and its expression counterparts. [*Example:*

```
&ac.operator int*i; // syntax error:
                    // parsed as: &(ac.operator int *)i
                    // not as: &(ac.operator int)*i
```

The `*` is the pointer declarator and not the multiplication operator. — *end example*] — *end note*]

1 Conversion functions are inherited.

2 Conversion functions can be virtual.

3 Conversion functions cannot be declared `static`.

12.807 Destructors [class.dtor]

1

A declaration of a destructor uses a function declarator (8.360.1) of the form

ptr-declarator (*parameter-declaration-clause*) *exception-specification*$_{opt}$ *attribute-specifier-seq*$_{opt}$

where the *ptr-declarator* consists solely of an *id-expression*, an optional *attribute-specifier-seq*, and optional surrounding parentheses, and the *id-expression* has one of the following forms:

— in a *member-declaration* that belongs to the *member-specification* of a class but is not a friend declaration (11.94), the *id-expression* is `~`*class-name* and the *class-name* is the injected-class-name (Clause 9) of the immediately-enclosing class;

— in a *member-declaration* that belongs to the *member-specification* of a class template but is not a friend declaration, the *id-expression* is `~`*class-name* and the *class-name* names the current instantiation (14.1319.0.1) of the immediately-enclosing class template; or

— in a declaration at namespace scope or in a friend declaration, the *id-expression* is *nested-name-specifier* `~`*class-name* and the *class-name* names the same class as the *nested-name-specifier*.

The *class-name* shall not be a *typedef-name*. A destructor shall take no arguments (8.360.1). In a destructor declaration, each *decl-specifier* of the optional *decl-specifier-seq* shall be `friend`, `inline`, or `virtual`.

1

A destructor is used to destroy objects of its class type. The address of a destructor shall not be taken. A destructor can be invoked for a `const`, `volatile` or `const volatile` object. `const` and `volatile` semantics (7.567.0.1) are not applied on an object under destruction. They stop being in effect when the destructor for the most derived object (1.319) starts.

1

A declaration of a destructor that does not have an *exception-specification* is implicitly considered to have the same *exception-specification* as an implicit declaration (15.228).

1

If a class has no user-declared destructor, a destructor is implicitly declared as defaulted (8.706). An implicitly-declared destructor is an `inline public` member of its class.

1

A defaulted destructor for a class `X` is defined as deleted if:

— `X` is a union-like class that has a variant member with a non-trivial destructor,

— any potentially constructed subobject has class type `M` (or array thereof) and `M` has a deleted destructor or a destructor that is inaccessible from the defaulted destructor,

— or, for a virtual destructor, lookup of the non-array deallocation function results in an ambiguity or in a function that is deleted or inaccessible from the defaulted destructor.

A destructor is trivial if it is not user-provided and if:

— the destructor is not `virtual`,

— all of the direct base classes of its class have trivial destructors, and

— for all of the non-static data members of its class that are of class type (or array thereof), each such class has a trivial destructor.

Otherwise, the destructor is *non-trivial*.

1

A destructor that is defaulted and not defined as deleted is *implicitly defined* when it is odr-used (3.91) to destroy an object of its class type (3.2235) or when it is explicitly defaulted after its first declaration.

1 Before the defaulted destructor for a class is implicitly defined, all the non-user-provided destructors for its base classes and its non-static data members shall have been implicitly defined.

2

After executing the body of the destructor and destroying any automatic objects allocated within the body, a destructor for class `X` calls the destructors for `X`'s direct non-variant non-static data members, the destructors for `X`'s direct base classes and, if `X` is the type of the most derived class (12.1026.1), its destructor calls the destructors for `X`'s virtual base classes. All destructors are called as if they were referenced with a qualified name, that is, ignoring any possible virtual overriding destructors in more derived classes. Bases and members are destroyed in the reverse order of the completion of their constructor (see 12.1026.1). A `return` statement (6.270.2) in a destructor might not directly return to the caller; before transferring control to the caller, the destructors for the members and bases are called. Destructors for elements of an array are called in reverse order of their construction (see 12.1011).

1

A destructor can be declared `virtual` (10.113) or pure `virtual` (10.276); if any objects of that class or any derived class are created in the program, the destructor shall be defined. If a class has a base class with a virtual destructor, its destructor (whether user- or implicitly-declared) is virtual.

1 [*Note:* some language constructs have special semantics when used during destruction; see 12.1483. — *end note*]

2

A destructor is invoked implicitly

— for a constructed object with static storage duration (3.2242.1) at program termination (3.2200.1),

— for a constructed object with thread storage duration (3.2291.1) at thread exit,

— for a constructed object with automatic storage duration (3.2301.1) when the block in which an object is created exits (6.419),

— for a constructed temporary object when its lifetime ends (12.306).

In each case, the context of the invocation is the context of the construction of the object. A destructor is also invoked implicitly through use of a *delete-expression* (5.2455.1) for a constructed object allocated by a *new-expression* (5.2221.1); the context of the invocation is the *delete-expression*. [*Note:* An array of class type contains several subobjects for each of which the destructor is invoked. — *end note*] A destructor can also be invoked explicitly. A destructor is *potentially invoked* if it is invoked or as specified in 5.2221.1 and 12.1026.1. A program is ill-formed if a destructor that is potentially invoked is deleted or not accessible from the context of the invocation.

1

At the point of definition of a virtual destructor (including an implicit definition (12.1577)), the non-array deallocation function is looked up in the scope of the destructor's class (10.52), and, if no declaration is found, the function is looked up in the global scope. If the result of this lookup is ambiguous or inaccessible, or if the lookup selects a placement deallocation function or a function with a deleted definition (8.706), the program is ill-formed. [*Note:* This assures that a deallocation function corresponding to the dynamic type of an object is available for the *delete-expression* (12.988). — *end note*]

1

In an explicit destructor call, the destructor name appears as a `~` followed by a *type-name* or *decltype-specifier* that denotes the destructor's class type. The invocation of a destructor is subject to the usual rules for member functions (9.401); that is, if the object is not of the destructor's class type and not of a class derived from the destructor's class type (including when the destructor is invoked via a null pointer value),

the program has undefined behavior. [*Note:* invoking `delete` on a null pointer does not call the destructor; see 5.2455.1. — *end note*] [*Example:*

```
struct B {
  virtual ~B() { }
};
struct D : B {
  ~D() { }
};

D D_object;
typedef B B_alias;
B* B_ptr = &D_object;

void f() {
  D_object.B::~B();              // calls B's destructor
  B_ptr->~B();                   // calls D's destructor
  B_ptr->~B_alias();             // calls D's destructor
  B_ptr->B_alias::~B();          // calls B's destructor
  B_ptr->B_alias::~B_alias();    // calls B's destructor
}
```

— *end example*] [*Note:* An explicit destructor call must always be written using a member access operator (5.1194.1) or a qualified-id (5.398); in particular, the *unary-expression* `~X()` in a member function is not an explicit destructor call (5.1898.1). — *end note*]

1

[*Note:* explicit calls of destructors are rarely needed. One use of such calls is for objects placed at specific addresses using a *new-expression* with the placement option. Such use of explicit placement and destruction of objects can be necessary to cope with dedicated hardware resources and for writing memory management facilities. For example,

```
void* operator new(std::size_t, void* p) { return p; }
struct X {
  X(int);
  ~X();
};
void f(X* p);

void g() {                       // rare, specialized use:
  char* buf = new char[sizeof(X)];
  X* p = new(buf) X(222);        // use buf[] and initialize
  f(p);
  p->X::~X();                    // cleanup
}
```

— *end note*]

1

Once a destructor is invoked for an object, the object no longer exists; the behavior is undefined if the destructor is invoked for an object whose lifetime has ended (3.2563). [*Example:* if the destructor for an automatic object is explicitly invoked, and the block is subsequently left in a manner that would ordinarily invoke implicit destruction of the object, the behavior is undefined. — *end example*]

1

[*Note:* the notation for explicit call of a destructor can be used for any scalar type name (5.1181.1). Allowing this makes it possible to write code without having to know if a destructor exists for a given type. For example,

```
typedef int I;
I* p;
p->I::~I();
```

— *end note*]

12.988 Free store [class.free]

1

Any allocation function for a class `T` is a static member (even if not explicitly declared `static`).

1 [*Example:*

```
class Arena;
struct B {
  void* operator new(std::size_t, Arena*);
};
struct D1 : B {
};

Arena*  ap;
void foo(int i) {
  new (ap) D1;       // calls B::operator new(std::size_t, Arena*)
  new D1[i];         // calls ::operator new[](std::size_t)
  new D1;            // ill-formed: ::operator new(std::size_t) hidden
}
```

— *end example*]

2

When an object is deleted with a *delete-expression* (5.2455.1), a *deallocation function* (`operator delete()` for non-array objects or `operator delete[]()` for arrays) is (implicitly) called to reclaim the storage occupied by the object (3.2428.0.1).

1

Class-specific deallocation function lookup is a part of general deallocation function lookup (5.2455.1) and occurs as follows. If the *delete-expression* is used to deallocate a class object whose static type has a virtual destructor, the deallocation function is the one selected at the point of definition of the dynamic type's virtual destructor (12.807).[119] Otherwise, if the *delete-expression* is used to deallocate an object of class `T` or array thereof, the static and dynamic types of the object shall be identical and the deallocation function's name is looked up in the scope of `T`. If this lookup fails to find the name, general deallocation function lookup (5.2455.1) continues. If the result of the lookup is ambiguous or inaccessible, or if the lookup selects a placement deallocation function, the program is ill-formed.

1

Any deallocation function for a class `X` is a static member (even if not explicitly declared `static`). [*Example:*

```
class X {
  void operator delete(void*);
  void operator delete[](void*, std::size_t);
```

119) A similar provision is not needed for the array version of `operator delete` because 5.2455.1 requires that in this situation, the static type of the object to be deleted be the same as its dynamic type.

```
};

class Y {
  void operator delete(void*, std::size_t);
  void operator delete[](void*);
};
```

— *end example*]

1

Since member allocation and deallocation functions are `static` they cannot be virtual. [*Note:* however, when the *cast-expression* of a *delete-expression* refers to an object of class type, because the deallocation function actually called is looked up in the scope of the class that is the dynamic type of the object, if the destructor is virtual, the effect is the same. For example,

```
struct B {
  virtual ~B();
  void operator delete(void*, std::size_t);
};

struct D : B {
  void operator delete(void*);
};

void f() {
  B* bp = new D;
  delete bp;        //1: uses D::operator delete(void*)
}
```

Here, storage for the non-array object of class `D` is deallocated by `D::operator delete()`, due to the virtual destructor. — *end note*] [*Note:* Virtual destructors have no effect on the deallocation function actually called when the *cast-expression* of a *delete-expression* refers to an array of objects of class type. For example,

```
struct B {
  virtual ~B();
  void operator delete[](void*, std::size_t);
};

struct D : B {
  void operator delete[](void*, std::size_t);
};

void f(int i) {
  D* dp = new D[i];
  delete [] dp;     // uses D::operator delete[](void*, std::size_t)
  B* bp = new D[i];
  delete[] bp;      // undefined behavior
}
```

— *end note*]

1 Access to the deallocation function is checked statically. Hence, even though a different one might actually be executed, the statically visible deallocation function is required to be accessible. [*Example:* for the call on line //1 above, if `B::operator delete()` had been `private`, the delete expression would have been ill-formed. — *end example*]

2 [*Note:* If a deallocation function has no explicit *exception-specification*, it is treated as if it were specified with `noexcept(true)` (15.228). — *end note*]

12.1011 Initialization [class.init]

1

When no initializer is specified for an object of (possibly cv-qualified) class type (or array thereof), or the initializer has the form `()`, the object is initialized as specified in 8.821.

1

An object of class type (or array thereof) can be explicitly initialized; see 12.1021.1 and 12.1026.1.

1

When an array of class objects is initialized (either explicitly or implicitly) and the elements are initialized by constructor, the constructor shall be called for each element of the array, following the subscript order; see 8.251.1. [*Note:* Destructors for the array elements are called in reverse order of their construction. — *end note*]

12.1021.1 Explicit initialization [class.expl.init]

1

An object of class type can be initialized with a parenthesized *expression-list*, where the *expression-list* is construed as an argument list for a constructor that is called to initialize the object. Alternatively, a single *assignment-expression* can be specified as an *initializer* using the `=` form of initialization. Either direct-initialization semantics or copy-initialization semantics apply; see 8.821. [*Example:*

```
struct complex {
  complex();
  complex(double);
  complex(double,double);
};

complex sqrt(complex,complex);

complex a(1);                   // initialize by a call of
                                // complex(double)
complex b = a;                  // initialize by a copy of a
complex c = complex(1,2);       // construct complex(1,2)
                                // using complex(double,double)
                                // copy/move it into c
complex d = sqrt(b,c);          // call sqrt(complex,complex)
                                // and copy/move the result into d
complex e;                      // initialize by a call of
                                // complex()
complex f = 3;                  // construct complex(3) using
                                // complex(double)
                                // copy/move it into f
complex g = { 1, 2 };           // initialize by a call of
                                // complex(double, double)
```

— *end example*] [*Note:* overloading of the assignment operator (13.1178.1) has no effect on initialization. — *end note*]

1

An object of class type can also be initialized by a *braced-init-list*. List-initialization semantics apply; see 8.821 and 8.2435.1. [*Example:*

```
complex v[6] = { 1, complex(1,2), complex(), 2 };
```

Here, `complex::complex(double)` is called for the initialization of `v[0]` and `v[3]`, `complex::complex(double, double)` is called for the initialization of `v[1]`, `complex::complex()` is called for the initialization `v[2]`, `v[4]`, and `v[5]`. For another example,

```
struct X {
  int i;
  float f;
  complex c;
} x = { 99, 88.8, 77.7 };
```

Here, `x.i` is initialized with 99, `x.f` is initialized with 88.8, and `complex::complex(double)` is called for the initialization of `x.c`. — *end example*] [*Note:* Braces can be elided in the *initializer-list* for any aggregate, even if the aggregate has members of a class type with user-defined type conversions; see 8.1817.1. — *end note*]

1

[*Note:* If `T` is a class type with no default constructor, any declaration of an object of type `T` (or array thereof) is ill-formed if no *initializer* is explicitly specified (see 12.1011 and 8.821). — *end note*]

1 [*Note:* the order in which objects with static or thread storage duration are initialized is described in 3.1937.1 and 6.419. — *end note*]

12.1026.1 Initializing bases and members [class.base.init]

1

In the definition of a constructor for a class, initializers for direct and virtual base subobjects and non-static data members can be specified by a *ctor-initializer*, which has the form

ctor-initializer:
: *mem-initializer-list*
mem-initializer-list:
mem-initializer ...$_{opt}$
mem-initializer ...$_{opt}$, *mem-initializer-list*
mem-initializer:
mem-initializer-id (*expression-list*$_{opt}$)
mem-initializer-id braced-init-list
mem-initializer-id:
class-or-decltype
identifier

1

In a *mem-initializer-id* an initial unqualified *identifier* is looked up in the scope of the constructor's class and, if not found in that scope, it is looked up in the scope containing the constructor's definition. [*Note:* If the constructor's class contains a member with the same name as a direct or virtual base class of the class, a *mem-initializer-id* naming the member or base class and composed of a single identifier refers to the class member. A *mem-initializer-id* for the hidden base class may be specified using a qualified name. — *end note*] Unless the *mem-initializer-id* names the constructor's class, a non-static data member of the constructor's class, or a direct or virtual base of that class, the *mem-initializer* is ill-formed.

1

A *mem-initializer-list* can initialize a base class using any *class-or-decltype* that denotes that base class type. [*Example:*

```
struct A { A(); };
  typedef A global_A;
  struct B { };
  struct C: public A, public B { C(); };
  C::C(): global_A() { }        // mem-initializer for base A
```

— *end example*]

1 If a *mem-initializer-id* is ambiguous because it designates both a direct non-virtual base class and an inherited virtual base class, the *mem-initializer* is ill-formed. [*Example:*

```
struct A { A(); };
struct B: public virtual A { };
struct C: public A, public B { C(); };
C::C(): A() { }                 // ill-formed: which A?
```

— *end example*]

2

A *ctor-initializer* may initialize a variant member of the constructor's class. If a *ctor-initializer* specifies more than one *mem-initializer* for the same member or for the same base class, the *ctor-initializer* is ill-formed.

1

A *mem-initializer-list* can delegate to another constructor of the constructor's class using any *class-or-decltype* that denotes the constructor's class itself. If a *mem-initializer-id* designates the constructor's class, it shall be the only *mem-initializer*; the constructor is a *delegating constructor*, and the constructor selected by the *mem-initializer* is the *target constructor*. The *principal constructor* is the first constructor invoked in the construction of an object (that is, not a target constructor for that object's construction). The target constructor is selected by overload resolution. Once the target constructor returns, the body of the delegating constructor is executed. If a constructor delegates to itself directly or indirectly, the program is ill-formed; no diagnostic is required. [*Example:*

```
struct C {
  C( int ) { }                  // #1: non-delegating constructor
  C(): C(42) { }                // #2: delegates to #1
  C( char c ) : C(42.0) { }     // #3: ill-formed due to recursion with #4
  C( double d ) : C('a') { }    // #4: ill-formed due to recursion with #3
};
```

— *end example*]

1

The *expression-list* or *braced-init-list* in a *mem-initializer* is used to initialize the designated subobject (or, in the case of a delegating constructor, the complete class object) according to the initialization rules of 8.821 for direct-initialization.

[*Example:*

```
struct B1 { B1(int); /* ... */ };
struct B2 { B2(int); /* ... */ };
struct D : B1, B2 {
  D(int);
  B1 b;
  const int c;
};

D::D(int a) : B2(a+1), B1(a+2), c(a+3), b(a+4)
```

```
  { /* ... */ }
D d(10);
```

— *end example*] The initialization performed by each *mem-initializer* constitutes a full-expression. Any expression in a *mem-initializer* is evaluated as part of the full-expression that performs the initialization. A *mem-initializer* where the *mem-initializer-id* denotes a virtual base class is ignored during execution of a constructor of any class that is not the most derived class.

1

In a non-delegating constructor, if a given potentially constructed subobject is not designated by a *mem-initializer-id* (including the case where there is no *mem-initializer-list* because the constructor has no *ctor-initializer*), then

— if the entity is a non-static data member that has a *brace-or-equal-initializer* and either
  — the constructor's class is a union (9.815), and no other variant member of that union is designated by a *mem-initializer-id* or
  — the constructor's class is not a union, and, if the entity is a member of an anonymous union, no other member of that union is designated by a *mem-initializer-id*,

  the entity is initialized as specified in 8.821;

— otherwise, if the entity is an anonymous union or a variant member (9.815), no initialization is performed;

— otherwise, the entity is default-initialized (8.821).

[*Note:* An abstract class (10.276) is never a most derived class, thus its constructors never initialize virtual base classes, therefore the corresponding *mem-initializer*s may be omitted. — *end note*] An attempt to initialize more than one non-static data member of a union renders the program ill-formed. [*Note:* After the call to a constructor for class `X` for an object with automatic or dynamic storage duration has completed, if the constructor was not invoked as part of value-initialization and a member of `X` is neither initialized nor given a value during execution of the *compound-statement* of the body of the constructor, the member has an indeterminate value. — *end note*] [*Example:*

```
struct A {
  A();
};

struct B {
  B(int);
};

struct C {
  C() { }                 // initializes members as follows:
  A a;                      // OK: calls A::A()
  const B b;                // error: B has no default constructor
  int i;                    // OK: i has indeterminate value
  int j = 5;                // OK: j has the value 5
};
```

— *end example*]

1

If a given non-static data member has both a *brace-or-equal-initializer* and a *mem-initializer*, the initialization specified by the *mem-initializer* is performed, and the non-static data member's *brace-or-equal-initializer* is ignored. [*Example:* Given

```
struct A {
  int i = /* some integer expression with side effects */ ;
  A(int arg) : i(arg) { }
  // ...
};
```

the `A(int)` constructor will simply initialize `i` to the value of `arg`, and the side effects in `i`'s *brace-or-equal-initializer* will not take place. — *end example*]

1

In a non-delegating constructor, the destructor for each potentially constructed subobject of class type is potentially invoked (12.807). [*Note:* This provision ensures that destructors can be called for fully-constructed sub-objects in case an exception is thrown (15.89). — *end note*]

1

In a non-delegating constructor, initialization proceeds in the following order:

— First, and only for the constructor of the most derived class (1.319), virtual base classes are initialized in the order they appear on a depth-first left-to-right traversal of the directed acyclic graph of base classes, where "left-to-right" is the order of appearance of the base classes in the derived class *base-specifier-list*.

— Then, direct base classes are initialized in declaration order as they appear in the *base-specifier-list* (regardless of the order of the *mem-initializers*).

— Then, non-static data members are initialized in the order they were declared in the class definition (again regardless of the order of the *mem-initializers*).

— Finally, the *compound-statement* of the constructor body is executed.

[*Note:* The declaration order is mandated to ensure that base and member subobjects are destroyed in the reverse order of initialization. — *end note*]

1 [*Example:*

```
struct V {
  V();
  V(int);
};

struct A : virtual V {
  A();
  A(int);
};

struct B : virtual V {
  B();
  B(int);
};

struct C : A, B, virtual V {
  C();
  C(int);
};

A::A(int i) : V(i) { /* ... */ }
```

```
B::B(int i) { /* ... */ }
C::C(int i) { /* ... */ }

V v(1);             // use V(int)
A a(2);             // use V(int)
B b(3);             // use V()
C c(4);             // use V()
```

— *end example*]

2

Names in the *expression-list* or *braced-init-list* of a *mem-initializer* are evaluated in the scope of the constructor for which the *mem-initializer* is specified. [*Example:*

```
class X {
  int a;
  int b;
  int i;
  int j;
public:
  const int& r;
  X(int i): r(a), b(i), i(i), j(this->i) { }
};
```

initializes `X::r` to refer to `X::a`, initializes `X::b` with the value of the constructor parameter `i`, initializes `X::i` with the value of the constructor parameter `i`, and initializes `X::j` with the value of `X::i`; this takes place each time an object of class `X` is created. — *end example*] [*Note:* Because the *mem-initializer* are evaluated in the scope of the constructor, the `this` pointer can be used in the *expression-list* of a *mem-initializer* to refer to the object being initialized. — *end note*]

1

Member functions (including virtual member functions, 10.113) can be called for an object under construction. Similarly, an object under construction can be the operand of the `typeid` operator (5.1393.1) or of a `dynamic_cast` (5.1295.1). However, if these operations are performed in a *ctor-initializer* (or in a function called directly or indirectly from a *ctor-initializer*) before all the *mem-initializer*s for base classes have completed, the result of the operation is undefined. [*Example:*

```
class A {
public:
  A(int);
};

class B : public A {
  int j;
public:
  int f();
  B() : A(f()),       // undefined: calls member function
                      // but base A not yet initialized
  j(f()) { }          // well-defined: bases are all initialized
};

class C {
public:
  C(int);
};
```

```
class D : public B, C {
  int i;
public:
  D() : C(f()),       // undefined: calls member function
                      // but base C not yet initialized
  i(f()) { }          // well-defined: bases are all initialized
};
```

— *end example*]

1 [*Note:* 12.1483 describes the result of virtual function calls, `typeid` and `dynamic_cast`s during construction for the well-defined cases; that is, describes the *polymorphic behavior* of an object under construction. — *end note*]

2 A *mem-initializer* followed by an ellipsis is a pack expansion (14.703.1) that initializes the base classes specified by a pack expansion in the *base-specifier-list* for the class. [*Example:*

```
template<class... Mixins>
class X : public Mixins... {
public:
  X(const Mixins&... mixins) : Mixins(mixins)... { }
};
```

— *end example*]

12.1483 Construction and destruction [class.cdtor]

1

For an object with a non-trivial constructor, referring to any non-static member or base class of the object before the constructor begins execution results in undefined behavior. For an object with a non-trivial destructor, referring to any non-static member or base class of the object after the destructor finishes execution results in undefined behavior. [*Example:*

```
struct X { int i; };
struct Y : X { Y(); };                  // non-trivial
struct A { int a; };
struct B : public A { int j; Y y; };    // non-trivial

extern B bobj;
B* pb = &bobj;                          // OK
int* p1 = &bobj.a;                      // undefined, refers to base class member
int* p2 = &bobj.y.i;                    // undefined, refers to member's member

A* pa = &bobj;                          // undefined, upcast to a base class type
B bobj;                                 // definition of bobj

extern X xobj;
int* p3 = &xobj.i;                      //OK, X is a trivial class
X xobj;
```

1 For another example,

```
struct W { int j; };
struct X : public virtual W { };
struct Y {
  int* p;
  X x;
```

```
  Y() : p(&x.j) {    // undefined, x is not yet constructed
    }
};
```

— *end example*]

2

To explicitly or implicitly convert a pointer (a glvalue) referring to an object of class `X` to a pointer (reference) to a direct or indirect base class `B` of `X`, the construction of `X` and the construction of all of its direct or indirect bases that directly or indirectly derive from `B` shall have started and the destruction of these classes shall not have completed, otherwise the conversion results in undefined behavior. To form a pointer to (or access the value of) a direct non-static member of an object `obj`, the construction of `obj` shall have started and its destruction shall not have completed, otherwise the computation of the pointer value (or accessing the member value) results in undefined behavior. [*Example:*

```
struct A { };
struct B : virtual A { };
struct C : B { };
struct D : virtual A { D(A*); };
struct X { X(A*); };

struct E : C, D, X {
  E() : D(this),     // undefined: upcast from E* to A*
                     // might use path E* → D* → A*
                     // but D is not constructed
                     // D((C*)this), // defined:
                     // E* → C* defined because E() has started
                     // and C* → A* defined because
                     // C fully constructed
  X(this) {          // defined: upon construction of X,
                     // C/B/D/A sublattice is fully constructed
  }
};
```

— *end example*]

1

Member functions, including virtual functions (10.113), can be called during construction or destruction (12.1026.1). When a virtual function is called directly or indirectly from a constructor or from a destructor, including during the construction or destruction of the class's non-static data members, and the object to which the call applies is the object (call it `x`) under construction or destruction, the function called is the final overrider in the constructor's or destructor's class and not one overriding it in a more-derived class. If the virtual function call uses an explicit class member access (5.1194.1) and the object expression refers to the complete object of `x` or one of that object's base class subobjects but not `x` or one of its base class subobjects, the behavior is undefined. [*Example:*

```
struct V {
  virtual void f();
  virtual void g();
};

struct A : virtual V {
  virtual void f();
};
```

```
struct B : virtual V {
  virtual void g();
  B(V*, A*);
};

struct D : A, B {
  virtual void f();
  virtual void g();
  D() : B((A*)this, this) { }
};

B::B(V* v, A* a) {
  f();              // calls V::f, not A::f
  g();              // calls B::g, not D::g
  v->g();           // v is base of B, the call is well-defined, calls B::g
  a->f();           // undefined behavior, a's type not a base of B
}
```

— *end example*]

1

The `typeid` operator (5.1393.1) can be used during construction or destruction (12.1026.1). When `typeid` is used in a constructor (including the *mem-initializer* or *brace-or-equal-initializer* for a non-static data member) or in a destructor, or used in a function called (directly or indirectly) from a constructor or destructor, if the operand of `typeid` refers to the object under construction or destruction, `typeid` yields the `std::type_info` object representing the constructor or destructor's class. If the operand of `typeid` refers to the object under construction or destruction and the static type of the operand is neither the constructor or destructor's class nor one of its bases, the result of `typeid` is undefined.

1

`dynamic_cast`s (5.1295.1) can be used during construction or destruction (12.1026.1). When a `dynamic_cast` is used in a constructor (including the *mem-initializer* or *brace-or-equal-initializer* for a non-static data member) or in a destructor, or used in a function called (directly or indirectly) from a constructor or destructor, if the operand of the `dynamic_cast` refers to the object under construction or destruction, this object is considered to be a most derived object that has the type of the constructor or destructor's class. If the operand of the `dynamic_cast` refers to the object under construction or destruction and the static type of the operand is not a pointer to or object of the constructor or destructor's own class or one of its bases, the `dynamic_cast` results in undefined behavior.

[*Example:*

```
struct V {
  virtual void f();
};

struct A : virtual V { };

struct B : virtual V {
  B(V*, A*);
};

struct D : A, B {
  D() : B((A*)this, this) { }
};
```

```
B::B(V* v, A* a) {
  typeid(*this);                // type_info for B
  typeid(*v);                   // well-defined: *v has type V, a base of B
                                // yields type_info for B
  typeid(*a);                   // undefined behavior: type A not a base of B
  dynamic_cast<B*>(v);          // well-defined: v of type V*, V base of B
                                // results in B*
  dynamic_cast<B*>(a);          // undefined behavior,
                                // a has type A*, A not a base of B
}
```

— *end example*]

12.1577 Copying and moving class objects [class.copy]

1

A class object can be copied or moved in two ways: by initialization (12.30, 8.821), including for function argument passing (5.955.1) and for function value return (6.270.2); and by assignment (5.3483). Conceptually, these two operations are implemented by a copy/move constructor (12.30) and copy/move assignment operator (13.1178.1).

1

A non-template constructor for class `X` is a copy constructor if its first parameter is of type `X&`, `const X&`, `volatile X&` or `const volatile X&`, and either there are no other parameters or else all other parameters have default arguments (8.545.1). [*Example:* `X::X(const X&)` and `X::X(X&,int=1)` are copy constructors.

```
struct X {
  X(int);
  X(const X&, int = 1);
};
X a(1);             // calls X(int);
X b(a, 0);          // calls X(const X&, int);
X c = b;            // calls X(const X&, int);
```

— *end example*]

1

A non-template constructor for class `X` is a move constructor if its first parameter is of type `X&&`, `const X&&`, `volatile X&&`, or `const volatile X&&`, and either there are no other parameters or else all other parameters have default arguments (8.545.1). [*Example:* `Y::Y(Y&&)` is a move constructor.

```
struct Y {
  Y(const Y&);
  Y(Y&&);
};
extern Y f(int);
Y d(f(1));          // calls Y(Y&&)
Y e = d;            // calls Y(const Y&)
```

— *end example*]

1 [*Note:* All forms of copy/move constructor may be declared for a class. [*Example:*

```
struct X {
  X(const X&);
  X(X&);            // OK
  X(X&&);
```

```
  X(const X&&);       // OK, but possibly not sensible
};
```

— *end example*] — *end note*]

2 [*Note:* If a class `X` only has a copy constructor with a parameter of type `X&`, an initializer of type `const X` or `volatile X` cannot initialize an object of type (possibly cv-qualified) `X`. [*Example:*

```
struct X {
  X();                // default constructor
  X(X&);              // copy constructor with a nonconst parameter
};
const X cx;
X x = cx;             // error: X::X(X&) cannot copy cx into x
```

— *end example*] — *end note*]

3

A declaration of a constructor for a class `X` is ill-formed if its first parameter is of type (optionally cv-qualified) `X` and either there are no other parameters or else all other parameters have default arguments. A member function template is never instantiated to produce such a constructor signature. [*Example:*

```
struct S {
  template<typename T> S(T);
  S();
};

S g;

void h() {
  S a(g);             // does not instantiate the member template to produce S::S<S>(S);
                      // uses the implicitly declared copy constructor
}
```

— *end example*]

1

If the class definition does not explicitly declare a copy constructor, one is declared *implicitly*. If the class definition declares a move constructor or move assignment operator, the implicitly declared copy constructor is defined as deleted; otherwise, it is defined as defaulted (8.706). The latter case is deprecated if the class has a user-declared copy assignment operator or a user-declared destructor.

1

The implicitly-declared copy constructor for a class `X` will have the form

```
X::X(const X&)
```

if each potentially constructed subobject of a class type `M` (or array thereof) has a copy constructor whose first parameter is of type `const M&` or `const volatile M&`.[120] Otherwise, the implicitly-declared copy constructor will have the form

```
X::X(X&)
```

1

120) This implies that the reference parameter of the implicitly-declared copy constructor cannot bind to a `volatile` lvalue; see C.1.9.

If the definition of a class `X` does not explicitly declare a move constructor, one will be implicitly declared as defaulted if and only if

— `X` does not have a user-declared copy constructor,

— `X` does not have a user-declared copy assignment operator,

— `X` does not have a user-declared move assignment operator, and

— `X` does not have a user-declared destructor.

[*Note:* When the move constructor is not implicitly declared or explicitly supplied, expressions that otherwise would have invoked the move constructor may instead invoke a copy constructor. — *end note*]

1

The implicitly-declared move constructor for class `X` will have the form

```
X::X(X&&)
```

1

An implicitly-declared copy/move constructor is an `inline public` member of its class. A defaulted copy/move constructor for a class `X` is defined as deleted (8.793.1) if `X` has:

— a variant member with a non-trivial corresponding constructor and `X` is a union-like class,

— a potentially constructed subobject type `M` (or array thereof) that cannot be copied/moved because overload resolution (13.88), as applied to `M`'s corresponding constructor, results in an ambiguity or a function that is deleted or inaccessible from the defaulted constructor,

— any potentially constructed subobject of a type with a destructor that is deleted or inaccessible from the defaulted constructor, or,

— for the copy constructor, a non-static data member of rvalue reference type.

A defaulted move constructor that is defined as deleted is ignored by overload resolution (13.88, 13.1084). [*Note:* A deleted move constructor would otherwise interfere with initialization from an rvalue which can use the copy constructor instead. — *end note*]

1

A copy/move constructor for class `X` is trivial if it is not user-provided, its parameter-type-list is equivalent to the parameter-type-list of an implicit declaration, and if

— class `X` has no virtual functions (10.113) and no virtual base classes (10.31), and

— class `X` has no non-static data members of volatile-qualified type, and

— the constructor selected to copy/move each direct base class subobject is trivial, and

— for each non-static data member of `X` that is of class type (or array thereof), the constructor selected to copy/move that member is trivial;

otherwise the copy/move constructor is *non-trivial*.

1

A copy/move constructor that is defaulted and not defined as deleted is *implicitly defined* if it is odr-used (3.91) or when it is explicitly defaulted after its first declaration. [*Note:* The copy/move constructor is

implicitly defined even if the implementation elided its odr-use (3.91, 12.306). — *end note*] If the implicitly-defined constructor would satisfy the requirements of a `constexpr` constructor (7.328.1), the implicitly-defined constructor is `constexpr`.

1 Before the defaulted copy/move constructor for a class is implicitly defined, all non-user-provided copy/-move constructors for its potentially constructed subobjects shall have been implicitly defined. [*Note:* An implicitly-declared copy/move constructor has an *exception-specification* (15.228). — *end note*]

2

The implicitly-defined copy/move constructor for a non-union class `X` performs a memberwise copy/move of its bases and members. [*Note: brace-or-equal-initializer*s of non-static data members are ignored. See also the example in 12.1026.1. — *end note*] The order of initialization is the same as the order of initialization of bases and members in a user-defined constructor (see 12.1026.1). Let `x` be either the parameter of the constructor or, for the move constructor, an xvalue referring to the parameter. Each base or non-static data member is copied/moved in the manner appropriate to its type:

— if the member is an array, each element is direct-initialized with the corresponding subobject of `x`;

— if a member `m` has rvalue reference type `T&&`, it is direct-initialized with `static_cast<T&&>(x.m)`;

— otherwise, the base or member is direct-initialized with the corresponding base or member of `x`.

Virtual base class subobjects shall be initialized only once by the implicitly-defined copy/move constructor (see 12.1026.1).

1

The implicitly-defined copy/move constructor for a union `X` copies the object representation (3.2955) of `X`.

1

A user-declared *copy* assignment operator `X::operator=` is a non-static non-template member function of class `X` with exactly one parameter of type `X`, `X&`, `const X&`, `volatile X&` or `const volatile X&`.[121] [*Note:* An overloaded assignment operator must be declared to have only one parameter; see 13.1178.1. — *end note*] [*Note:* More than one form of copy assignment operator may be declared for a class. — *end note*] [*Note:* If a class `X` only has a copy assignment operator with a parameter of type `X&`, an expression of type const `X` cannot be assigned to an object of type `X`. [*Example:*

```
struct X {
  X();
  X& operator=(X&);
};
const X cx;
X x;
void f() {
  x = cx;            // error: X::operator=(X&) cannot assign cx into x
}
```

— *end example*] — *end note*]

1

If the class definition does not explicitly declare a copy assignment operator, one is declared *implicitly*. If the class definition declares a move constructor or move assignment operator, the implicitly declared copy assignment operator is defined as deleted; otherwise, it is defined as defaulted (8.706). The latter case is

121) Because a template assignment operator or an assignment operator taking an rvalue reference parameter is never a copy assignment operator, the presence of such an assignment operator does not suppress the implicit declaration of a copy assignment operator. Such assignment operators participate in overload resolution with other assignment operators, including copy assignment operators, and, if selected, will be used to assign an object.

deprecated if the class has a user-declared copy constructor or a user-declared destructor. The implicitly-declared copy assignment operator for a class `X` will have the form

```
X& X::operator=(const X&)
```

if

— each direct base class `B` of `X` has a copy assignment operator whose parameter is of type `const B&`, `const volatile B&` or `B`, and

— for all the non-static data members of `X` that are of a class type `M` (or array thereof), each such class type has a copy assignment operator whose parameter is of type `const M&`, `const volatile M&` or `M`.[122]

Otherwise, the implicitly-declared copy assignment operator will have the form

```
X& X::operator=(X&)
```

1

A user-declared move assignment operator `X::operator=` is a non-static non-template member function of class `X` with exactly one parameter of type `X&&`, `const X&&`, `volatile X&&`, or `const volatile X&&`. [*Note:* An overloaded assignment operator must be declared to have only one parameter; see 13.1178.1. — *end note*] [*Note:* More than one form of move assignment operator may be declared for a class. — *end note*]

1

If the definition of a class `X` does not explicitly declare a move assignment operator, one will be implicitly declared as defaulted if and only if

— `X` does not have a user-declared copy constructor,

— `X` does not have a user-declared move constructor,

— `X` does not have a user-declared copy assignment operator, and

— `X` does not have a user-declared destructor.

[*Example:* The class definition

```
struct S {
  int a;
  S& operator=(const S&) = default;
};
```

will not have a default move assignment operator implicitly declared because the copy assignment operator has been user-declared. The move assignment operator may be explicitly defaulted.

```
struct S {
  int a;
  S& operator=(const S&) = default;
  S& operator=(S&&) = default;
};
```

— *end example*]

1 The implicitly-declared move assignment operator for a class `X` will have the form

```
X& X::operator=(X&&);
```

122) This implies that the reference parameter of the implicitly-declared copy assignment operator cannot bind to a `volatile` lvalue; see C.1.9.

2 The implicitly-declared copy/move assignment operator for class `X` has the return type `X&`; it returns the object for which the assignment operator is invoked, that is, the object assigned to. An implicitly-declared copy/move assignment operator is an `inline public` member of its class.

3

A defaulted copy/move assignment operator for class `X` is defined as deleted if `X` has:

— a variant member with a non-trivial corresponding assignment operator and `X` is a union-like class, or

— a non-static data member of `const` non-class type (or array thereof), or

— a non-static data member of reference type, or

— a potentially constructed subobject of class type `M` (or array thereof) that cannot be copied/moved because overload resolution (13.88), as applied to `M`'s corresponding assignment operator, results in an ambiguity or a function that is deleted or inaccessible from the defaulted assignment operator.

A defaulted move assignment operator that is defined as deleted is ignored by overload resolution (13.88, 13.1084).

1

Because a copy/move assignment operator is implicitly declared for a class if not declared by the user, a base class copy/move assignment operator is always hidden by the corresponding assignment operator of a derived class (13.1178.1). A *using-declaration* (7.1311.1) that brings in from a base class an assignment operator with a parameter type that could be that of a copy/move assignment operator for the derived class is not considered an explicit declaration of such an operator and does not suppress the implicit declaration of the derived class operator; the operator introduced by the *using-declaration* is hidden by the implicitly-declared operator in the derived class.

1

A copy/move assignment operator for class `X` is trivial if it is not user-provided, its parameter-type-list is equivalent to the parameter-type-list of an implicit declaration, and if

— class `X` has no virtual functions (10.113) and no virtual base classes (10.31), and

— class `X` has no non-static data members of volatile-qualified type, and

— the assignment operator selected to copy/move each direct base class subobject is trivial, and

— for each non-static data member of `X` that is of class type (or array thereof), the assignment operator selected to copy/move that member is trivial;

otherwise the copy/move assignment operator is *non-trivial*.

1 A copy/move assignment operator for a class `X` that is defaulted and not defined as deleted is *implicitly defined* when it is odr-used (3.91) (e.g., when it is selected by overload resolution to assign to an object of its class type) or when it is explicitly defaulted after its first declaration. The implicitly-defined copy/move assignment operator is `constexpr` if

(1.1) — `X` is a literal type, and

(1.2) — the assignment operator selected to copy/move each direct base class subobject is a `constexpr` function, and

(1.3) — for each non-static data member of `X` that is of class type (or array thereof), the assignment operator selected to copy/move that member is a `constexpr` function.

2 Before the defaulted copy/move assignment operator for a class is implicitly defined, all non-user-provided copy/move assignment operators for its direct base classes and its non-static data members shall have been implicitly defined. [*Note:* An implicitly-declared copy/move assignment operator has an *exception-specification* (15.228). — *end note*]

3

The implicitly-defined copy/move assignment operator for a non-union class `X` performs memberwise copy-/move assignment of its subobjects. The direct base classes of `X` are assigned first, in the order of their declaration in the *base-specifier-list*, and then the immediate non-static data members of `X` are assigned, in the order in which they were declared in the class definition. Let `x` be either the parameter of the function or, for the move operator, an xvalue referring to the parameter. Each subobject is assigned in the manner appropriate to its type:

— if the subobject is of class type, as if by a call to `operator=` with the subobject as the object expression and the corresponding subobject of `x` as a single function argument (as if by explicit qualification; that is, ignoring any possible virtual overriding functions in more derived classes);

— if the subobject is an array, each element is assigned, in the manner appropriate to the element type;

— if the subobject is of scalar type, the built-in assignment operator is used.

It is unspecified whether subobjects representing virtual base classes are assigned more than once by the implicitly-defined copy assignment operator. [*Example:*

```
struct V { };
struct A : virtual V { };
struct B : virtual V { };
struct C : B, A { };
```

It is unspecified whether the virtual base class subobject `V` is assigned twice by the implicitly-defined copy assignment operator for `C`. — *end example*] [*Note:* This does not apply to move assignment, as a defaulted move assignment operator is deleted if the class has virtual bases. — *end note*]

1

The implicitly-defined copy assignment operator for a union `X` copies the object representation (3.2955) of `X`.

1

A program is ill-formed if the copy/move constructor or the copy/move assignment operator for an object is implicitly odr-used and the special member function is not accessible (Clause 11). [*Note:* Copying/moving one object into another using the copy/move constructor or the copy/move assignment operator does not change the layout or size of either object. — *end note*]

1

When certain criteria are met, an implementation is allowed to omit the copy/move construction of a class object, even if the constructor selected for the copy/move operation and/or the destructor for the object have side effects. In such cases, the implementation treats the source and target of the omitted copy/move operation as simply two different ways of referring to the same object, and the destruction of that object occurs at the later of the times when the two objects would have been destroyed without the optimization.[123] This elision of copy/move operations, called *copy elision*, is permitted in the following circumstances (which may be combined to eliminate multiple copies):

123) Because only one object is destroyed instead of two, and one copy/move constructor is not executed, there is still one object destroyed for each one constructed.

— in a `return` statement in a function with a class return type, when the expression is the name of a non-volatile automatic object (other than a function or catch-clause parameter) with the same cv-unqualified type as the function return type, the copy/move operation can be omitted by constructing the automatic object directly into the function's return value

— in a *throw-expression*, when the operand is the name of a non-volatile automatic object (other than a function or catch-clause parameter) whose scope does not extend beyond the end of the innermost enclosing *try-block* (if there is one), the copy/move operation from the operand to the exception object (15.7) can be omitted by constructing the automatic object directly into the exception object

— when a temporary class object that has not been bound to a reference (12.306) would be copied/moved to a class object with the same cv-unqualified type, the copy/move operation can be omitted by constructing the temporary object directly into the target of the omitted copy/move

— when the *exception-declaration* of an exception handler (Clause 15) declares an object of the same type (except for cv-qualification) as the exception object (15.7), the copy operation can be omitted by treating the *exception-declaration* as an alias for the exception object if the meaning of the program will be unchanged except for the execution of constructors and destructors for the object declared by the *exception-declaration*. [*Note:* There cannot be a move from the exception object because it is always an lvalue. — *end note*]

[*Example:*

```
class Thing {
public:
  Thing();
  ~Thing();
  Thing(const Thing&);
};

Thing f() {
  Thing t;
  return t;
}

Thing t2 = f();
```

Here the criteria for elision can be combined to eliminate two calls to the copy constructor of class `Thing`: the copying of the local automatic object `t` into the temporary object for the return value of function `f()` and the copying of that temporary object into object `t2`. Effectively, the construction of the local object `t` can be viewed as directly initializing the global object `t2`, and that object's destruction will occur at program exit. Adding a move constructor to `Thing` has the same effect, but it is the move construction from the temporary object to `t2` that is elided. — *end example*]

1

When the criteria for elision of a copy/move operation are met, but not for an *exception-declaration*, and the object to be copied is designated by an lvalue, or when the *expression* in a `return` statement is a (possibly parenthesized) *id-expression* that names an object with automatic storage duration declared in the body or *parameter-declaration-clause* of the innermost enclosing function or *lambda-expression*, overload resolution to select the constructor for the copy is first performed as if the object were designated by an rvalue. If the first overload resolution fails or was not performed, or if the type of the first parameter of the selected constructor is not an rvalue reference to the object's type (possibly cv-qualified), overload resolution is performed again, considering the object as an lvalue. [*Note:* This two-stage overload resolution must be performed regardless of whether copy elision will occur. It determines the constructor to be called if elision is not performed, and the selected constructor must be accessible even if the call is elided. — *end note*]

[*Example:*

```
class Thing {
public:
  Thing();
  ~Thing();
  Thing(Thing&&);
private:
  Thing(const Thing&);
};

Thing f(bool b) {
  Thing t;
  if (b)
    throw t;                    // OK: Thing(Thing&&) used (or elided) to throw t
  return t;                     // OK: Thing(Thing&&) used (or elided) to return t
}

Thing t2 = f(false);            // OK: Thing(Thing&&) used (or elided) to construct t2
```

— *end example*]

12.2400 Inheriting constructors [class.inhctor]

1

A *using-declaration* (7.1311.1) that names a constructor implicitly declares a set of *inheriting constructors*. The *candidate set of inherited constructors* from the class `X` named in the *using-declaration* consists of actual constructors and notional constructors that result from the transformation of defaulted parameters as follows:

— all non-template constructors of `X`, and

— for each non-template constructor of `X` that has at least one parameter with a default argument, the set of constructors that results from omitting any ellipsis parameter specification and successively omitting parameters with a default argument from the end of the parameter-type-list, and

— all constructor templates of `X`, and

— for each constructor template of `X` that has at least one parameter with a default argument, the set of constructor templates that results from omitting any ellipsis parameter specification and successively omitting parameters with a default argument from the end of the parameter-type-list.

1 The *constructor characteristics* of a constructor or constructor template are

(1.1) — the template parameter list (14.97), if any,

(1.2) — the *parameter-type-list* (8.360.1),

(1.3) — absence or presence of `explicit` (12.708.1), and

(1.4) — absence or presence of `constexpr` (7.328.1).

2

For each non-template constructor in the candidate set of inherited constructors other than a constructor having no parameters or a copy/move constructor having a single parameter, a constructor is implicitly declared with the same constructor characteristics unless there is a user-declared constructor with the same signature in the complete class where the *using-declaration* appears or the constructor would be a default,

copy, or move constructor for that class. Similarly, for each constructor template in the candidate set of inherited constructors, a constructor template is implicitly declared with the same constructor characteristics unless there is an equivalent user-declared constructor template (14.975.0.1) in the complete class where the *using-declaration* appears. [*Note:* Default arguments are not inherited. An *exception-specification* is implied as specified in 15.228. — *end note*]

1

A constructor so declared has the same access as the corresponding constructor in `X`. It is deleted if the corresponding constructor in `X` is deleted (8.706). An inheriting constructor shall not be explicitly instantiated (14.1841.1) or explicitly specialized (14.1921.1).

1

[*Note:* Default and copy/move constructors may be implicitly declared as specified in 12.30 and 12.1577. — *end note*]

1 [*Example:*

```
struct B1 {
  B1(int);
};

struct B2 {
  B2(int = 13, int = 42);
};

struct D1 : B1 {
  using B1::B1;
};

struct D2 : B2 {
  using B2::B2;
};
```

The candidate set of inherited constructors in `D1` for `B1` is

— `B1(const B1&)`

— `B1(B1&&)`

— `B1(int)`

The set of constructors present in `D1` is

— `D1()`, implicitly-declared default constructor, ill-formed if odr-used

— `D1(const D1&)`, implicitly-declared copy constructor, not inherited

— `D1(D1&&)`, implicitly-declared move constructor, not inherited

— `D1(int)`, implicitly-declared inheriting constructor

The candidate set of inherited constructors in `D2` for `B2` is

— `B2(const B2&)`

— `B2(B2&&)`

— `B2(int = 13, int = 42)`

— `B2(int = 13)`

— `B2()`

The set of constructors present in `D2` is

— `D2()`, implicitly-declared default constructor, not inherited

— `D2(const D2&)`, implicitly-declared copy constructor, not inherited

— `D2(D2&&)`, implicitly-declared move constructor, not inherited

— `D2(int, int)`, implicitly-declared inheriting constructor

— `D2(int)`, implicitly-declared inheriting constructor

— *end example*]

1

[*Note:* If two *using-declaration*s declare inheriting constructors with the same signatures, the program is ill-formed (9.194, 13.12), because an implicitly-declared constructor introduced by the first *using-declaration* is not a user-declared constructor and thus does not preclude another declaration of a constructor with the same signature by a subsequent *using-declaration*. [*Example:*

```
struct B1 {
  B1(int);
};

struct B2 {
  B2(int);
};

struct D1 : B1, B2 {
  using B1::B1;
  using B2::B2;
};                    // ill-formed: attempts to declare D1(int) twice

struct D2 : B1, B2 {
  using B1::B1;
  using B2::B2;
  D2(int);            // OK: user declaration supersedes both implicit declarations
};
```

— *end example*] — *end note*]

1

An inheriting constructor for a class is implicitly defined when it is odr-used (3.91) to create an object of its class type (1.319). An implicitly-defined inheriting constructor performs the set of initializations of the class that would be performed by a user-written `inline` constructor for that class with a *mem-initializer-list* whose only *mem-initializer* has a *mem-initializer-id* that names the base class denoted in the *nested-name-specifier* of the *using-declaration* and an *expression-list* as specified below, and where the *compound-statement* in its function body is empty (12.1026.1). If that user-written constructor would be ill-formed, the program is ill-formed. Each *expression* in the *expression-list* is of the form `static_cast<T&&>(p)`, where `p` is the name of the corresponding constructor parameter and `T` is the declared type of `p`.

1

[*Example:*

```
struct B1 {
  B1(int) { }
};

struct B2 {
  B2(double) { }
};

struct D1 : B1 {
  using B1::B1;      // implicitly declares D1(int)
  int x;
};

void test() {
  D1 d(6);           // OK: d.x is not initialized
  D1 e;              // error: D1 has no default constructor
}

struct D2 : B2 {
  using B2::B2;      // OK: implicitly declares D2(double)
  B1 b;
};

D2 f(1.0);           // error: B1 has no default constructor

template< class T >
struct D : T {
  using T::T;        // declares all constructors from class T
  ~D() { std::clog << "Destroying wrapper" << std::endl; }
};
```

Class template `D` wraps any class and forwards all of its constructors, while writing a message to the standard log whenever an object of class `D` is destroyed. — *end example*]

13 Overloading [over]

1

When two or more different declarations are specified for a single name in the same scope, that name is said to be *overloaded*. By extension, two declarations in the same scope that declare the same name but with different types are called *overloaded declarations*. Only function and function template declarations can be overloaded; variable and type declarations cannot be overloaded.

1

When an overloaded function name is used in a call, which overloaded function declaration is being referenced is determined by comparing the types of the arguments at the point of use with the types of the parameters in the overloaded declarations that are visible at the point of use. This function selection process is called *overload resolution* and is defined in 13.88. [*Example:*

```
double abs(double);
int abs(int);

abs(1);             // calls abs(int);
abs(1.0);           // calls abs(double);
```

— *end example*]

13.12 Overloadable declarations [over.load]

1

Not all function declarations can be overloaded. Those that cannot be overloaded are specified here. A program is ill-formed if it contains two such non-overloadable declarations in the same scope. [*Note:* This restriction applies to explicit declarations in a scope, and between such declarations and declarations made through a *using-declaration* (7.1311.1). It does not apply to sets of functions fabricated as a result of name lookup (e.g., because of *using-directives*) or overload resolution (e.g., for operator functions). — *end note*]

1

Certain function declarations cannot be overloaded:

— Function declarations that differ only in the return type cannot be overloaded.

— Member function declarations with the same name and the same *parameter-type-list* cannot be overloaded if any of them is a `static` member function declaration (9.559). Likewise, member function template declarations with the same name, the same *parameter-type-list*, and the same template parameter lists cannot be overloaded if any of them is a `static` member function template declaration. The types of the implicit object parameters constructed for the member functions for the purpose of overload resolution (13.133.1) are not considered when comparing parameter-type-lists for enforcement of this rule. In contrast, if there is no `static` member function declaration among a set of member function declarations with the same name and the same parameter-type-list, then these member function declarations can be overloaded if they differ in the type of their implicit object parameter. [*Example:* the following illustrates this distinction:

```
class X {
  static void f();
  void f();                    // ill-formed
  void f() const;              // ill-formed
```

```
  void f() const volatile;      // ill-formed
  void g();
  void g() const;               // OK: no static g
  void g() const volatile;      // OK: no static g
};
```

— *end example*]

— Member function declarations with the same name and the same *parameter-type-list* as well as member function template declarations with the same name, the same *parameter-type-list*, and the same template parameter lists cannot be overloaded if any of them, but not all, have a *ref-qualifier* (8.360.1). [*Example:*

```
class Y {
  void h() &;
  void h() const &;             // OK
  void h() &&;                  // OK, all declarations have a ref-qualifier
  void i() &;
  void i() const;               // ill-formed, prior declaration of i
                                // has a ref-qualifier
};
```

— *end example*]

1

[*Note:* As specified in 8.360.1, function declarations that have equivalent parameter declarations declare the same function and therefore cannot be overloaded:

— Parameter declarations that differ only in the use of equivalent typedef "types" are equivalent. A `typedef` is not a separate type, but only a synonym for another type (7.235.1). [*Example:*

```
typedef int Int;

void f(int i);
void f(Int i);                  // OK: redeclaration of f(int)
void f(int i) { /* ...  */ }
void f(Int i) { /* ...  */ }    // error: redefinition of f(int)
```

— *end example*]

Enumerations, on the other hand, are distinct types and can be used to distinguish overloaded function declarations. [*Example:*

```
enum E { a };

void f(int i) { /* ...  */ }
void f(E i)   { /* ...  */ }
```

— *end example*]

— Parameter declarations that differ only in a pointer `*` versus an array `[]` are equivalent. That is, the array declaration is adjusted to become a pointer declaration (8.360.1). Only the second and subsequent array dimensions are significant in parameter types (8.251.1). [*Example:*

```
int f(char*);
int f(char[]);                  // same as f(char*);
int f(char[7]);                 // same as f(char*);
int f(char[9]);                 // same as f(char*);
```

```
int g(char(*)[10]);
int g(char[5][10]);              // same as g(char(*)[10]);
int g(char[7][10]);              // same as g(char(*)[10]);
int g(char(*)[20]);              // different from g(char(*)[10]);
```

— *end example*]

— Parameter declarations that differ only in that one is a function type and the other is a pointer to the same function type are equivalent. That is, the function type is adjusted to become a pointer to function type (8.360.1). [*Example:*

```
void h(int());
void h(int (*)());               // redeclaration of h(int())
void h(int x()) { }              // definition of h(int())
void h(int (*x)()) { }           // ill-formed: redefinition of h(int())
```

— *end example*]

— Parameter declarations that differ only in the presence or absence of `const` and/or `volatile` are equivalent. That is, the `const` and `volatile` type-specifiers for each parameter type are ignored when determining which function is being declared, defined, or called. [*Example:*

```
typedef const int cInt;

int f (int);
int f (const int);               // redeclaration of f(int)
int f (int) { /* ...  */ }       // definition of f(int)
int f (cInt) { /* ...  */ }      // error: redefinition of f(int)
```

— *end example*]

Only the `const` and `volatile` type-specifiers at the outermost level of the parameter type specification are ignored in this fashion; `const` and `volatile` type-specifiers buried within a parameter type specification are significant and can be used to distinguish overloaded function declarations.[124] In particular, for any type `T`, "pointer to `T`," "pointer to `const T`," and "pointer to `volatile T`" are considered distinct parameter types, as are "reference to `T`," "reference to `const T`," and "reference to `volatile T`."

— Two parameter declarations that differ only in their default arguments are equivalent. [*Example:* consider the following:

```
void f (int i, int j);
void f (int i, int j = 99);      // OK: redeclaration of f(int, int)
void f (int i = 88, int j);      // OK: redeclaration of f(int, int)
void f ();                       // OK: overloaded declaration of f

void prog () {
    f (1, 2);                    // OK: call f(int, int)
    f (1);                       // OK: call f(int, int)
    f ();                        // Error: f(int, int) or f()?
}
```

— *end example*] — *end note*]

124) When a parameter type includes a function type, such as in the case of a parameter type that is a pointer to function, the `const` and `volatile` type-specifiers at the outermost level of the parameter type specifications for the inner function type are also ignored.

13.85 Declaration matching [over.dcl]

1

Two function declarations of the same name refer to the same function if they are in the same scope and have equivalent parameter declarations (13.12). A function member of a derived class is *not* in the same scope as a function member of the same name in a base class. [*Example:*

```
struct B {
  int f(int);
};

struct D : B {
  int f(const char*);
};
```

Here `D::f(const char*)` hides `B::f(int)` rather than overloading it.

```
void h(D* pd) {
  pd->f(1);                    // error:
                               // D::f(const char*) hides B::f(int)
  pd->B::f(1);                 // OK
  pd->f("Ben");                // OK, calls D::f
}
```

— *end example*]

1 A locally declared function is not in the same scope as a function in a containing scope. [*Example:*

```
void f(const char*);
void g() {
  extern void f(int);
  f("asdf");                   // error: f(int) hides f(const char*)
                               // so there is no f(const char*) in this scope
}

void caller () {
  extern void callee(int, int);
  {
    extern void callee(int);   // hides callee(int, int)
    callee(88, 99);            // error: only callee(int) in scope
  }
}
```

— *end example*]

2 Different versions of an overloaded member function can be given different access rules. [*Example:*

```
class buffer {
private:
    char* p;
    int size;
protected:
    buffer(int s, char* store) { size = s; p = store; }
public:
    buffer(int s) { p = new char[size = s]; }
};
```

— *end example*]

13.88 Overload resolution [over.match]

1

Overload resolution is a mechanism for selecting the best function to call given a list of expressions that are to be the arguments of the call and a set of *candidate functions* that can be called based on the context of the call. The selection criteria for the best function are the number of arguments, how well the arguments match the parameter-type-list of the candidate function, how well (for non-static member functions) the object matches the implicit object parameter, and certain other properties of the candidate function. [*Note:* The function selected by overload resolution is not guaranteed to be appropriate for the context. Other restrictions, such as the accessibility of the function, can make its use in the calling context ill-formed. — *end note*]

1

Overload resolution selects the function to call in seven distinct contexts within the language:

— invocation of a function named in the function call syntax (13.230.0.1.1);

— invocation of a function call operator, a pointer-to-function conversion function, a reference-to-pointer-to-function conversion function, or a reference-to-function conversion function on a class object named in the function call syntax (13.250.0.0.1);

— invocation of the operator referenced in an expression (13.264.0.1);

— invocation of a constructor for direct-initialization (8.821) of a class object (13.354.0.1);

— invocation of a user-defined conversion for copy-initialization (8.821) of a class object (13.368.0.1);

— invocation of a conversion function for initialization of an object of a nonclass type from an expression of class type (13.396.0.1); and

— invocation of a conversion function for conversion to a glvalue or class prvalue to which a reference (8.2156.1) will be directly bound (13.413.0.1).

Each of these contexts defines the set of candidate functions and the list of arguments in its own unique way. But, once the candidate functions and argument lists have been identified, the selection of the best function is the same in all cases:

— First, a subset of the candidate functions (those that have the proper number of arguments and meet certain other conditions) is selected to form a set of viable functions (13.530.1).

— Then the best viable function is selected based on the implicit conversion sequences (13.779.0.1) needed to match each argument to the corresponding parameter of each viable function.

1

If a best viable function exists and is unique, overload resolution succeeds and produces it as the result. Otherwise overload resolution fails and the invocation is ill-formed. When overload resolution succeeds, and the best viable function is not accessible (Clause 11) in the context in which it is used, the program is ill-formed.

13.133.1 Candidate functions and argument lists [over.match.funcs]

1

The subclauses of 13.133.1 describe the set of candidate functions and the argument list submitted to overload resolution in each of the seven contexts in which overload resolution is used. The source transformations and constructions defined in these subclauses are only for the purpose of describing the overload resolution process. An implementation is not required to use such transformations and constructions.

1

The set of candidate functions can contain both member and non-member functions to be resolved against the same argument list. So that argument and parameter lists are comparable within this heterogeneous set, a member function is considered to have an extra parameter, called the *implicit object parameter*, which represents the object for which the member function has been called. For the purposes of overload resolution, both static and non-static member functions have an implicit object parameter, but constructors do not.

1

Similarly, when appropriate, the context can construct an argument list that contains an *implied object argument* to denote the object to be operated on. Since arguments and parameters are associated by position within their respective lists, the convention is that the implicit object parameter, if present, is always the first parameter and the implied object argument, if present, is always the first argument.

1

For non-static member functions, the type of the implicit object parameter is

— "lvalue reference to *cv* `X`" for functions declared without a *ref-qualifier* or with the `&` *ref-qualifier*

— "rvalue reference to *cv* `X`" for functions declared with the `&&` *ref-qualifier*

where `X` is the class of which the function is a member and *cv* is the cv-qualification on the member function declaration. [*Example:* for a `const` member function of class `X`, the extra parameter is assumed to have type "reference to `const X`". — *end example*] For conversion functions, the function is considered to be a member of the class of the implied object argument for the purpose of defining the type of the implicit object parameter. For non-conversion functions introduced by a *using-declaration* into a derived class, the function is considered to be a member of the derived class for the purpose of defining the type of the implicit object parameter. For static member functions, the implicit object parameter is considered to match any object (since if the function is selected, the object is discarded). [*Note:* No actual type is established for the implicit object parameter of a static member function, and no attempt will be made to determine a conversion sequence for that parameter (13.562.1). — *end note*]

1

During overload resolution, the implied object argument is indistinguishable from other arguments. The implicit object parameter, however, retains its identity since conversions on the corresponding argument shall obey these additional rules:

— no temporary object can be introduced to hold the argument for the implicit object parameter; and

— no user-defined conversions can be applied to achieve a type match with it.

For non-static member functions declared without a *ref-qualifier*, an additional rule applies:

— even if the implicit object parameter is not `const`-qualified, an rvalue can be bound to the parameter as long as in all other respects the argument can be converted to the type of the implicit object parameter. [*Note:* The fact that such an argument is an rvalue does not affect the ranking of implicit conversion sequences (13.946.0.1). — *end note*]

1 Because other than in list-initialization only one user-defined conversion is allowed in an implicit conversion sequence, special rules apply when selecting the best user-defined conversion (13.562.1, 13.779.0.1). [*Example:*

```
class T {
public:
  T();
};
```

```
class C : T {
public:
  C(int);
};
T a = 1;              // ill-formed: T(C(1)) not tried
```

— *end example*]

2

In each case where a candidate is a function template, candidate function template specializations are generated using template argument deduction (14.2662.1, 14.2114.1). Those candidates are then handled as candidate functions in the usual way.[125] A given name can refer to one or more function templates and also to a set of overloaded non-template functions. In such a case, the candidate functions generated from each function template are combined with the set of non-template candidate functions.

1 A defaulted move constructor or assignment operator (12.1577) that is defined as deleted is excluded from the set of candidate functions in all contexts.

13.230.0.1 Function call syntax [over.match.call]

1 In a function call (5.955.1)

postfix-expression `(` *expression-list*$_{opt}$`)`

if the *postfix-expression* denotes a set of overloaded functions and/or function templates, overload resolution is applied as specified in 13.230.0.1.1. If the *postfix-expression* denotes an object of class type, overload resolution is applied as specified in 13.250.0.0.1.

2 If the *postfix-expression* denotes the address of a set of overloaded functions and/or function templates, overload resolution is applied using that set as described above. If the function selected by overload resolution is a non-static member function, the program is ill-formed. [*Note:* The resolution of the address of an overload set in other contexts is described in 13.1084. — *end note*]

13.230.0.1.1 Call to named function [over.call.func]

1 Of interest in 13.230.0.1.1 are only those function calls in which the *postfix-expression* ultimately contains a name that denotes one or more functions that might be called. Such a *postfix-expression*, perhaps nested arbitrarily deep in parentheses, has one of the following forms:

postfix-expression:
  postfix-expression `.` *id-expression*
  postfix-expression `->` *id-expression*
  primary-expression

These represent two syntactic subcategories of function calls: qualified function calls and unqualified function calls.

2

In qualified function calls, the name to be resolved is an *id-expression* and is preceded by an `->` or `.` operator. Since the construct `A->B` is generally equivalent to `(*A).B`, the rest of Clause 13 assumes, without loss of generality, that all member function calls have been normalized to the form that uses an object and the `.` operator. Furthermore, Clause 13 assumes that the *postfix-expression* that is the left operand of the `.` operator has type "*cv* `T`" where `T` denotes a class[126]. Under this assumption, the *id-expression* in the call

125) The process of argument deduction fully determines the parameter types of the function template specializations, i.e., the parameters of function template specializations contain no template parameter types. Therefore, except where specified otherwise, function template specializations and non-template functions (8.360.1) are treated equivalently for the remainder of overload resolution.

126) Note that cv-qualifiers on the type of objects are significant in overload resolution for both glvalue and class prvalue objects.

is looked up as a member function of `T` following the rules for looking up names in classes (10.52). The function declarations found by that lookup constitute the set of candidate functions. The argument list is the *expression-list* in the call augmented by the addition of the left operand of the `.` operator in the normalized member function call as the implied object argument (13.133.1).

1

In unqualified function calls, the name is not qualified by an `->` or `.` operator and has the more general form of a *primary-expression*. The name is looked up in the context of the function call following the normal rules for name lookup in function calls (3.923). The function declarations found by that lookup constitute the set of candidate functions. Because of the rules for name lookup, the set of candidate functions consists (1) entirely of non-member functions or (2) entirely of member functions of some class `T`. In case (1), the argument list is the same as the *expression-list* in the call. In case (2), the argument list is the *expression-list* in the call augmented by the addition of an implied object argument as in a qualified function call. If the keyword `this` (9.508.1) is in scope and refers to class `T`, or a derived class of `T`, then the implied object argument is `(*this)`. If the keyword `this` is not in scope or refers to another class, then a contrived object of type `T` becomes the implied object argument[127]. If the argument list is augmented by a contrived object and overload resolution selects one of the non-static member functions of `T`, the call is ill-formed.

13.250.0.0.1 Call to object of class type [over.call.object]

1

If the *primary-expression* `E` in the function call syntax evaluates to a class object of type "*cv* `T`", then the set of candidate functions includes at least the function call operators of `T`. The function call operators of `T` are obtained by ordinary lookup of the name `operator()` in the context of `(E).operator()`.

1

In addition, for each non-explicit conversion function declared in `T` of the form

> `operator` *conversion-type-id* `()` *cv-qualifier* *ref-qualifier*$_{opt}$ *exception-specification*$_{opt}$ *attribute-specifier-seq*$_{opt}$`;`

where *cv-qualifier* is the same cv-qualification as, or a greater cv-qualification than, *cv*, and where *conversion-type-id* denotes the type "pointer to function of `(P1,...,Pn)` returning `R`", or the type "reference to pointer to function of `(P1,...,Pn)` returning `R`", or the type "reference to function of `(P1,...,Pn)` returning `R`", a *surrogate call function* with the unique name *call-function* and having the form

> `R` *call-function* `(` *conversion-type-id* `F, P1 a1, ... ,Pn an) { return F (a1,... ,an); }`

is also considered as a candidate function. Similarly, surrogate call functions are added to the set of candidate functions for each non-explicit conversion function declared in a base class of `T` provided the function is not hidden within `T` by another intervening declaration[128].

1

If such a surrogate call function is selected by overload resolution, the corresponding conversion function will be called to convert `E` to the appropriate function pointer or reference, and the function will then be invoked with the arguments of the call. If the conversion function cannot be called (e.g., because of an ambiguity), the program is ill-formed.

1 The argument list submitted to overload resolution consists of the argument expressions present in the function call syntax preceded by the implied object argument `(E)`. [*Note:* When comparing the call against

127) An implied object argument must be contrived to correspond to the implicit object parameter attributed to member functions during overload resolution. It is not used in the call to the selected function. Since the member functions all have the same implicit object parameter, the contrived object will not be the cause to select or reject a function.

128) Note that this construction can yield candidate call functions that cannot be differentiated one from the other by overload resolution because they have identical declarations or differ only in their return type. The call will be ambiguous if overload resolution cannot select a match to the call that is uniquely better than such undifferentiable functions.

the function call operators, the implied object argument is compared against the implicit object parameter of the function call operator. When comparing the call against a surrogate call function, the implied object argument is compared against the first parameter of the surrogate call function. The conversion function from which the surrogate call function was derived will be used in the conversion sequence for that parameter since it converts the implied object argument to the appropriate function pointer or reference required by that first parameter. — *end note*] [*Example:*

```
int f1(int);
int f2(float);
typedef int (*fp1)(int);
typedef int (*fp2)(float);
struct A {
  operator fp1() { return f1; }
  operator fp2() { return f2; }
} a;
int i = a(1);        // calls f1 via pointer returned from
                     // conversion function
```

— *end example*]

13.264.0.1 Operators in expressions [over.match.oper]

1

If no operand of an operator in an expression has a type that is a class or an enumeration, the operator is assumed to be a built-in operator and interpreted according to Clause 5. [*Note:* Because `.`, `.*`, and `::` cannot be overloaded, these operators are always built-in operators interpreted according to Clause 5. `?:` cannot be overloaded, but the rules in this subclause are used to determine the conversions to be applied to the second and third operands when they have class or enumeration type (5.3337). — *end note*] [*Example:*

```
struct String {
  String (const String&);
  String (const char*);
  operator const char* ();
};
String operator + (const String&, const String&);

void f(void) {
 const char* p= "one" + "two";  // ill-formed because neither
                                // operand has class or enumeration type
 int I = 1 + 1;                 // Always evaluates to 2 even if
                                // class or enumeration types exist that
                                // would perform the operation.
}
```

— *end example*]

1

If either operand has a type that is a class or an enumeration, a user-defined operator function might be declared that implements this operator or a user-defined conversion can be necessary to convert the operand to a type that is appropriate for a built-in operator. In this case, overload resolution is used to determine which operator function or built-in operator is to be invoked to implement the operator. Therefore, the operator notation is first transformed to the equivalent function-call notation as summarized in Table 11 (where `@` denotes one of the operators covered in the specified subclause).

1

Table 11 — Relationship between operator and function call notation

Subclause	Expression	As member function	As non-member function
13.1166.1	@a	(a).operator@ ( )	operator@ (a)
13.1170.1	a@b	(a).operator@ (b)	operator@ (a, b)
13.1178.1	a=b	(a).operator= (b)	
13.1210.1	a[b]	(a).operator[](b)	
13.1217.1	a->	(a).operator-> ( )	
13.1232.1	a@	(a).operator@ (0)	operator@ (a, 0)

For a unary operator `@` with an operand of a type whose cv-unqualified version is `T1`, and for a binary operator `@` with a left operand of a type whose cv-unqualified version is `T1` and a right operand of a type whose cv-unqualified version is `T2`, three sets of candidate functions, designated *member candidates*, *non-member candidates* and *built-in candidates*, are constructed as follows:

— If `T1` is a complete class type or a class currently being defined, the set of member candidates is the result of the qualified lookup of `T1::operator@` (13.230.0.1.1); otherwise, the set of member candidates is empty.

— The set of non-member candidates is the result of the unqualified lookup of `operator@` in the context of the expression according to the usual rules for name lookup in unqualified function calls (3.1092.1) except that all member functions are ignored. However, if no operand has a class type, only those non-member functions in the lookup set that have a first parameter of type `T1` or "reference to (possibly cv-qualified) `T1`", when `T1` is an enumeration type, or (if there is a right operand) a second parameter of type `T2` or "reference to (possibly cv-qualified) `T2`", when `T2` is an enumeration type, are candidate functions.

— For the operator `,`, the unary operator `&`, or the operator `->`, the built-in candidates set is empty. For all other operators, the built-in candidates include all of the candidate operator functions defined in 13.1258 that, compared to the given operator,
  — have the same operator name, and
  — accept the same number of operands, and
  — accept operand types to which the given operand or operands can be converted according to 13.779.0.1, and
  — do not have the same parameter-type-list as any non-member candidate that is not a function template specialization.

1

For the built-in assignment operators, conversions of the left operand are restricted as follows:

— no temporaries are introduced to hold the left operand, and

— no user-defined conversions are applied to the left operand to achieve a type match with the left-most parameter of a built-in candidate.

1 For all other operators, no such restrictions apply.

2

The set of candidate functions for overload resolution is the union of the member candidates, the non-member candidates, and the built-in candidates. The argument list contains all of the operands of the operator. The best function from the set of candidate functions is selected according to 13.530.1 and 13.562.1.[129] [*Example:*

129) If the set of candidate functions is empty, overload resolution is unsuccessful.

```
struct A {
  operator int();
};
A operator+(const A&, const A&);
void m() {
  A a, b;
  a + b;            // operator+(a,b) chosen over int(a) + int(b)
}
```

— *end example*]

1

If a built-in candidate is selected by overload resolution, the operands of class type are converted to the types of the corresponding parameters of the selected operation function, except that the second standard conversion sequence of a user-defined conversion sequence (13.844.0.0.1) is not applied. Then the operator is treated as the corresponding built-in operator and interpreted according to Clause 5. [*Example:*

```
struct X {
  operator double();
};

struct Y {
  operator int*();
};

int *a = Y() + 100.0;       // error: pointer arithmetic requires integral operand
int *b = Y() + X();         // error: pointer arithmetic requires integral operand
```

— *end example*]

1

The second operand of operator `->` is ignored in selecting an `operator->` function, and is not an argument when the `operator->` function is called. When `operator->` returns, the operator `->` is applied to the value returned, with the original second operand.[130]

1

If the operator is the operator `,`, the unary operator `&`, or the operator `->`, and there are no viable functions, then the operator is assumed to be the built-in operator and interpreted according to Clause 5.

1

[*Note:* The lookup rules for operators in expressions are different than the lookup rules for operator function names in a function call, as shown in the following example:

```
struct A { };
void operator + (A, A);

struct B {
  void operator + (B);
  void f ();
};

A a;
```

130) If the value returned by the `operator->` function has class type, this may result in selecting and calling another `operator->` function. The process repeats until an `operator->` function returns a value of non-class type.

```
void B::f() {
  operator+ (a,a);          // error: global operator hidden by member
  a + a;                    // OK: calls global operator+
}
```

— *end note*]

13.354.0.1 Initialization by constructor [over.match.ctor]

1

When objects of class type are direct-initialized (8.821), or copy-initialized from an expression of the same or a derived class type (8.821), overload resolution selects the constructor. For direct-initialization, the candidate functions are all the constructors of the class of the object being initialized. For copy-initialization, the candidate functions are all the converting constructors (12.708.1) of that class. The argument list is the *expression-list* or *assignment-expression* of the *initializer*.

13.368.0.1 Copy-initialization of class by user-defined conversion [over.match.copy]

1

Under the conditions specified in 8.821, as part of a copy-initialization of an object of class type, a user-defined conversion can be invoked to convert an initializer expression to the type of the object being initialized. Overload resolution is used to select the user-defined conversion to be invoked. [*Note:* The conversion performed for indirect binding to a reference to a possibly cv-qualified class type is determined in terms of a corresponding non-reference copy-initialization. — *end note*] Assuming that "*cv1* `T`" is the type of the object being initialized, with `T` a class type, the candidate functions are selected as follows:

— The converting constructors (12.708.1) of `T` are candidate functions.

— When the type of the initializer expression is a class type "*cv* `S`", the non-explicit conversion functions of `S` and its base classes are considered. When initializing a temporary to be bound to the first parameter of a constructor that takes a reference to possibly *cv*-qualified `T` as its first argument, called with a single argument in the context of direct-initialization of an object of type "*cv2* `T`", explicit conversion functions are also considered. Those that are not hidden within `S` and yield a type whose cv-unqualified version is the same type as `T` or is a derived class thereof are candidate functions. Conversion functions that return "reference to `X`" return lvalues or xvalues, depending on the type of reference, of type `X` and are therefore considered to yield `X` for this process of selecting candidate functions.

1

In both cases, the argument list has one argument, which is the initializer expression. [*Note:* This argument will be compared against the first parameter of the constructors and against the implicit object parameter of the conversion functions. — *end note*]

13.396.0.1 Initialization by conversion function [over.match.conv]

1

Under the conditions specified in 8.821, as part of an initialization of an object of nonclass type, a conversion function can be invoked to convert an initializer expression of class type to the type of the object being initialized. Overload resolution is used to select the conversion function to be invoked. Assuming that "*cv1* `T`" is the type of the object being initialized, and "*cv* `S`" is the type of the initializer expression, with `S` a class type, the candidate functions are selected as follows:

— The conversion functions of `S` and its base classes are considered. Those non-explicit conversion functions that are not hidden within `S` and yield type `T` or a type that can be converted to type `T` via a standard conversion sequence (13.837.0.0.1) are candidate functions. For direct-initialization, those explicit conversion functions that are not hidden within `S` and yield type `T` or a type that can be

converted to type `T` with a qualification conversion (4.246) are also candidate functions. Conversion functions that return a cv-qualified type are considered to yield the cv-unqualified version of that type for this process of selecting candidate functions. Conversion functions that return "reference to *cv2* `X`" return lvalues or xvalues, depending on the type of reference, of type "*cv2* `X`" and are therefore considered to yield `X` for this process of selecting candidate functions.

1 The argument list has one argument, which is the initializer expression. [*Note:* This argument will be compared against the implicit object parameter of the conversion functions. — *end note*]

13.413.0.1 Initialization by conversion function for direct reference binding [over.match.ref]

1

Under the conditions specified in 8.2156.1, a reference can be bound directly to a glvalue or class prvalue that is the result of applying a conversion function to an initializer expression. Overload resolution is used to select the conversion function to be invoked. Assuming that "*cv1* `T`" is the underlying type of the reference being initialized, and "*cv* `S`" is the type of the initializer expression, with `S` a class type, the candidate functions are selected as follows:

— The conversion functions of `S` and its base classes are considered. Those non-explicit conversion functions that are not hidden within `S` and yield type "lvalue reference to *cv2* `T2`" (when initializing an lvalue reference or an rvalue reference to function) or "*cv2* `T2`" or "rvalue reference to *cv2* `T2`" (when initializing an rvalue reference or an lvalue reference to function), where "*cv1* `T`" is reference-compatible (8.2156.1) with "*cv2* `T2`", are candidate functions. For direct-initialization, those explicit conversion functions that are not hidden within `S` and yield type "lvalue reference to *cv2* `T2`" or "*cv2* `T2`" or "rvalue reference to *cv2* `T2`," respectively, where `T2` is the same type as `T` or can be converted to type `T` with a qualification conversion (4.246), are also candidate functions.

1 The argument list has one argument, which is the initializer expression. [*Note:* This argument will be compared against the implicit object parameter of the conversion functions. — *end note*]

13.425.0.1 Initialization by list-initialization [over.match.list]

1

When objects of non-aggregate class type `T` are list-initialized (8.2435.1), overload resolution selects the constructor in two phases:

— Initially, the candidate functions are the initializer-list constructors (8.2435.1) of the class `T` and the argument list consists of the initializer list as a single argument.

— If no viable initializer-list constructor is found, overload resolution is performed again, where the candidate functions are all the constructors of the class `T` and the argument list consists of the elements of the initializer list.

If the initializer list has no elements and `T` has a default constructor, the first phase is omitted. In copy-list-initialization, if an `explicit` constructor is chosen, the initialization is ill-formed. [*Note:* This differs from other situations (13.354.0.1, 13.368.0.1), where only converting constructors are considered for copy-initialization. This restriction only applies if this initialization is part of the final result of overload resolution. — *end note*]

13.530.1 Viable functions [over.match.viable]

1

From the set of candidate functions constructed for a given context (13.133.1), a set of viable functions is chosen, from which the best function will be selected by comparing argument conversion sequences for the

best fit (13.562.1). The selection of viable functions considers relationships between arguments and function parameters other than the ranking of conversion sequences.

1

First, to be a viable function, a candidate function shall have enough parameters to agree in number with the arguments in the list.

— If there are m arguments in the list, all candidate functions having exactly m parameters are viable.

— A candidate function having fewer than m parameters is viable only if it has an ellipsis in its parameter list (8.360.1). For the purposes of overload resolution, any argument for which there is no corresponding parameter is considered to "match the ellipsis" (13.868.0.0.1) .

— A candidate function having more than m parameters is viable only if the *(m+1)*-st parameter has a default argument (8.545.1).[131] For the purposes of overload resolution, the parameter list is truncated on the right, so that there are exactly m parameters.

1

Second, for `F` to be a viable function, there shall exist for each argument an *implicit conversion sequence* (13.779.0.1) that converts that argument to the corresponding parameter of `F`. If the parameter has reference type, the implicit conversion sequence includes the operation of binding the reference, and the fact that an lvalue reference to non-`const` cannot be bound to an rvalue and that an rvalue reference cannot be bound to an lvalue can affect the viability of the function (see 13.870.0.0.1).

13.562.1 Best viable function [over.match.best]

1

Define ICSi(`F`) as follows:

— if `F` is a static member function, ICS*1*(`F`) is defined such that ICS*1*(`F`) is neither better nor worse than ICS*1*(`G`) for any function `G`, and, symmetrically, ICS*1*(`G`) is neither better nor worse than ICS*1*(`F`)[132]; otherwise,

— let ICSi(`F`) denote the implicit conversion sequence that converts the i-th argument in the list to the type of the i-th parameter of viable function `F`. 13.779.0.1 defines the implicit conversion sequences and 13.946.0.1 defines what it means for one implicit conversion sequence to be a better conversion sequence or worse conversion sequence than another.

Given these definitions, a viable function `F1` is defined to be a *better* function than another viable function `F2` if for all arguments i, ICSi(`F1`) is not a worse conversion sequence than ICSi(`F2`), and then

— for some argument j, ICSj(`F1`) is a better conversion sequence than ICSj(`F2`), or, if not that,

— the context is an initialization by user-defined conversion (see 8.821, 13.396.0.1, and 13.413.0.1) and the standard conversion sequence from the return type of `F1` to the destination type (i.e., the type of the entity being initialized) is a better conversion sequence than the standard conversion sequence from the return type of `F2` to the destination type. [*Example:*

```
struct A {
  A();
  operator int();
```

131) According to 8.545.1, parameters following the *(m+1)*-st parameter must also have default arguments.

132) If a function is a static member function, this definition means that the first argument, the implied object argument, has no effect in the determination of whether the function is better or worse than any other function.

```
  operator double();
} a;
int i = a;                    // a.operator int() followed by no conversion
                              // is better than a.operator double() followed by
                              // a conversion to int
float x = a;                  // ambiguous: both possibilities require conversions,
                              // and neither is better than the other
```

— *end example*] or, if not that,

— the context is an initialization by conversion function for direct reference binding (13.413.0.1) of a reference to function type, the return type of `F1` is the same kind of reference (i.e. lvalue or rvalue) as the reference being initialized, and the return type of `F2` is not [*Example:*

```
template <class T> struct A {
  operator T&();        // #1
  operator T&&();       // #2
};
typedef int Fn();
A<Fn> a;
Fn& lf = a;             // calls #1
Fn&& rf = a;            // calls #2
```

— *end example*] or, if not that,

— `F1` is not a function template specialization and `F2` is a function template specialization, or, if not that,

— `F1` and `F2` are function template specializations, and the function template for `F1` is more specialized than the template for `F2` according to the partial ordering rules described in 14.992.0.1.

1

If there is exactly one viable function that is a better function than all other viable functions, then it is the one selected by overload resolution; otherwise the call is ill-formed[133].

[*Example:*

```
void Fcn(const int*,  short);
void Fcn(int*, int);

int i;
short s = 0;

void f() {
  Fcn(&i, s);                 // is ambiguous because
                              // &i → int* is better than &i → const int*
                              // but s → short is also better than s → int

  Fcn(&i, 1L);                // calls Fcn(int*, int), because
                              // &i → int* is better than &i → const int*
                              // and 1L → short and 1L → int are indistinguishable
```

133) The algorithm for selecting the best viable function is linear in the number of viable functions. Run a simple tournament to find a function `W` that is not worse than any opponent it faced. Although another function `F` that `W` did not face might be at least as good as `W`, `F` cannot be the best function because at some point in the tournament `F` encountered another function `G` such that `F` was not better than `G`. Hence, `W` is either the best function or there is no best function. So, make a second pass over the viable functions to verify that `W` is better than all other functions.

```
  Fcn(&i,'c');                    // calls Fcn(int*, int), because
                                  // &i → int* is better than &i → const int*
                                  // and c → int is better than c → short
}
```

— *end example*]

1 If the best viable function resolves to a function for which multiple declarations were found, and if at least two of these declarations — or the declarations they refer to in the case of *using-declarations* — specify a default argument that made the function viable, the program is ill-formed. [*Example:*

```
namespace A {
  extern "C" void f(int = 5);
}
namespace B {
  extern "C" void f(int = 5);
}

using A::f;
using B::f;

void use() {
  f(3);                           // OK, default argument was not used for viability
  f();                            // Error: found default argument twice
}
```

— *end example*]

13.779.0.1 Implicit conversion sequences [over.best.ics]

1

An *implicit conversion sequence* is a sequence of conversions used to convert an argument in a function call to the type of the corresponding parameter of the function being called. The sequence of conversions is an implicit conversion as defined in Clause 4, which means it is governed by the rules for initialization of an object or reference by a single expression (8.821, 8.2156.1).

1

Implicit conversion sequences are concerned only with the type, cv-qualification, and value category of the argument and how these are converted to match the corresponding properties of the parameter. Other properties, such as the lifetime, storage class, alignment, or accessibility of the argument and whether or not the argument is a bit-field are ignored. So, although an implicit conversion sequence can be defined for a given argument-parameter pair, the conversion from the argument to the parameter might still be ill-formed in the final analysis.

1

A well-formed implicit conversion sequence is one of the following forms:

— a *standard conversion sequence* (13.837.0.0.1),

— a *user-defined conversion sequence* (13.844.0.0.1), or

— an *ellipsis conversion sequence* (13.868.0.0.1).

1

However, if the target is

— the first parameter of a constructor or

— the implicit object parameter of a user-defined conversion function

and the constructor or user-defined conversion function is a candidate by

— 13.354.0.1, when the argument is the temporary in the second step of a class copy-initialization,

— 13.368.0.1, 13.396.0.1, or 13.413.0.1 (in all cases), or

— the second phase of 13.425.0.1 when the initializer list has exactly one element, and the target is the first parameter of a constructor of class `X`, and the conversion is to `X` or reference to (possibly cv-qualified) `X`,

user-defined conversion sequences are not considered. [*Note:* These rules prevent more than one user-defined conversion from being applied during overload resolution, thereby avoiding infinite recursion. — *end note*] [*Example:*

```
struct Y { Y(int); };
struct A { operator int(); };
Y y1 = A();  // error: A::operator int() is not a candidate

struct X { };
struct B { operator X(); };
B b;
X x({b});    // error: B::operator X() is not a candidate
```

— *end example*]

1

For the case where the parameter type is a reference, see 13.870.0.0.1.

1

When the parameter type is not a reference, the implicit conversion sequence models a copy-initialization of the parameter from the argument expression. The implicit conversion sequence is the one required to convert the argument expression to a prvalue of the type of the parameter. [*Note:* When the parameter has a class type, this is a conceptual conversion defined for the purposes of Clause 13; the actual initialization is defined in terms of constructors and is not a conversion. — *end note*] Any difference in top-level cv-qualification is subsumed by the initialization itself and does not constitute a conversion. [*Example:* a parameter of type `A` can be initialized from an argument of type `const A`. The implicit conversion sequence for that case is the identity sequence; it contains no "conversion" from `const A` to `A`. — *end example*] When the parameter has a class type and the argument expression has the same type, the implicit conversion sequence is an identity conversion. When the parameter has a class type and the argument expression has a derived class type, the implicit conversion sequence is a derived-to-base Conversion from the derived class to the base class. [*Note:* There is no such standard conversion; this derived-to-base Conversion exists only in the description of implicit conversion sequences. — *end note*] A derived-to-base Conversion has Conversion rank (13.837.0.0.1).

1 In all contexts, when converting to the implicit object parameter or when converting to the left operand of an assignment operation only standard conversion sequences that create no temporary object for the result are allowed.

2

If no conversions are required to match an argument to a parameter type, the implicit conversion sequence is the standard conversion sequence consisting of the identity conversion (13.837.0.0.1).

1

If no sequence of conversions can be found to convert an argument to a parameter type or the conversion is otherwise ill-formed, an implicit conversion sequence cannot be formed.

1

If several different sequences of conversions exist that each convert the argument to the parameter type, the implicit conversion sequence associated with the parameter is defined to be the unique conversion sequence designated the *ambiguous conversion sequence*. For the purpose of ranking implicit conversion sequences as described in 13.946.0.1, the ambiguous conversion sequence is treated as a user-defined sequence that is indistinguishable from any other user-defined conversion sequence[134]. If a function that uses the ambiguous conversion sequence is selected as the best viable function, the call will be ill-formed because the conversion of one of the arguments in the call is ambiguous.

1 The three forms of implicit conversion sequences mentioned above are defined in the following subclauses.

13.837.0.0.1 Standard conversion sequences [over.ics.scs]

1

Table 12 summarizes the conversions defined in Clause 4 and partitions them into four disjoint categories: Lvalue Transformation, Qualification Adjustment, Promotion, and Conversion. [*Note:* These categories are orthogonal with respect to value category, cv-qualification, and data representation: the Lvalue Transformations do not change the cv-qualification or data representation of the type; the Qualification Adjustments do not change the value category or data representation of the type; and the Promotions and Conversions do not change the value category or cv-qualification of the type. — *end note*]

1

[*Note:* As described in Clause 4, a standard conversion sequence is either the Identity conversion by itself (that is, no conversion) or consists of one to three conversions from the other four categories. At most one conversion from each category is allowed in a single standard conversion sequence. If there are two or more conversions in the sequence, the conversions are applied in the canonical order: **Lvalue Transformation**, **Promotion** or **Conversion**, **Qualification Adjustment**. — *end note*]

1

Each conversion in Table 12 also has an associated rank (Exact Match, Promotion, or Conversion). These are used to rank standard conversion sequences (13.946.0.1). The rank of a conversion sequence is determined by considering the rank of each conversion in the sequence and the rank of any reference binding (13.870.0.0.1).

---

134) The ambiguous conversion sequence is ranked with user-defined conversion sequences because multiple conversion sequences for an argument can exist only if they involve different user-defined conversions. The ambiguous conversion sequence is indistinguishable from any other user-defined conversion sequence because it represents at least two user-defined conversion sequences, each with a different user-defined conversion, and any other user-defined conversion sequence must be indistinguishable from at least one of them.

This rule prevents a function from becoming non-viable because of an ambiguous conversion sequence for one of its parameters. Consider this example,

```
class B;
class A { A (B&);};
class B { operator A (); };
class C { C (B&); };
void f(A) { }
void f(C) { }
B b;
f(b);                          // ambiguous because b → C via constructor and
                               // b → A via constructor or conversion function.
```

If it were not for this rule, `f(A)` would be eliminated as a viable function for the call `f(b)` causing overload resolution to select `f(C)` as the function to call even though it is not clearly the best choice. On the other hand, if an `f(B)` were to be declared then `f(b)` would resolve to that `f(B)` because the exact match with `f(B)` is better than any of the sequences required to match `f(A)`.

If any of those has Conversion rank, the sequence has Conversion rank; otherwise, if any of those has Promotion rank, the sequence has Promotion rank; otherwise, the sequence has Exact Match rank.

Table 12 — Conversions

Conversion	Category	Rank	Subclause
No conversions required	Identity	Exact Match	
Lvalue-to-rvalue conversion	Lvalue Transformation		4.127
Array-to-pointer conversion			4.180
Function-to-pointer conversion			4.220
Qualification conversions	Qualification Adjustment		4.246
Integral promotions	Promotion	Promotion	4.262
Floating point promotion			4.389
Integral conversions	Conversion	Conversion	4.397
Floating point conversions			4.516
Floating-integral conversions			4.532
Pointer conversions			4.579
Pointer to member conversions			4.710
Boolean conversions			4.732

13.844.0.0.1 User-defined conversion sequences [over.ics.user]

1

A user-defined conversion sequence consists of an initial standard conversion sequence followed by a user-defined conversion (12.659) followed by a second standard conversion sequence. If the user-defined conversion is specified by a constructor (12.708.1), the initial standard conversion sequence converts the source type to the type required by the argument of the constructor. If the user-defined conversion is specified by a conversion function (12.742.1), the initial standard conversion sequence converts the source type to the implicit object parameter of the conversion function.

1

The second standard conversion sequence converts the result of the user-defined conversion to the target type for the sequence. Since an implicit conversion sequence is an initialization, the special rules for initialization by user-defined conversion apply when selecting the best user-defined conversion for a user-defined conversion sequence (see 13.562.1 and 13.779.0.1).

1

If the user-defined conversion is specified by a specialization of a conversion function template, the second standard conversion sequence shall have exact match rank.

1

A conversion of an expression of class type to the same class type is given Exact Match rank, and a conversion of an expression of class type to a base class of that type is given Conversion rank, in spite of the fact that a constructor (i.e., a user-defined conversion function) is called for those cases.

13.868.0.0.1 Ellipsis conversion sequences [over.ics.ellipsis]

1

An ellipsis conversion sequence occurs when an argument in a function call is matched with the ellipsis parameter specification of the function called (see 5.955.1).

13.870.0.0.1 Reference binding [over.ics.ref]

1

When a parameter of reference type binds directly (8.2156.1) to an argument expression, the implicit conversion sequence is the identity conversion, unless the argument expression has a type that is a derived class of the parameter type, in which case the implicit conversion sequence is a derived-to-base Conversion (13.779.0.1). [*Example:*

```
struct A {};
struct B : public A {} b;
int f(A&);
int f(B&);
int i = f(b);                    // calls f(B&), an exact match, rather than
                                 // f(A&), a conversion
```

— *end example*] If the parameter binds directly to the result of applying a conversion function to the argument expression, the implicit conversion sequence is a user-defined conversion sequence (13.844.0.0.1), with the second standard conversion sequence either an identity conversion or, if the conversion function returns an entity of a type that is a derived class of the parameter type, a derived-to-base Conversion.

1

When a parameter of reference type is not bound directly to an argument expression, the conversion sequence is the one required to convert the argument expression to the underlying type of the reference according to 13.779.0.1. Conceptually, this conversion sequence corresponds to copy-initializing a temporary of the underlying type with the argument expression. Any difference in top-level cv-qualification is subsumed by the initialization itself and does not constitute a conversion.

1

Except for an implicit object parameter, for which see 13.133.1, a standard conversion sequence cannot be formed if it requires binding an lvalue reference other than a reference to a non-volatile `const` type to an rvalue or binding an rvalue reference to an lvalue other than a function lvalue. [*Note:* This means, for example, that a candidate function cannot be a viable function if it has a non-`const` lvalue reference parameter (other than the implicit object parameter) and the corresponding argument is a temporary or would require one to be created to initialize the lvalue reference (see 8.2156.1). — *end note*]

1

Other restrictions on binding a reference to a particular argument that are not based on the types of the reference and the argument do not affect the formation of a standard conversion sequence, however. [*Example:* a function with an "lvalue reference to `int`" parameter can be a viable candidate even if the corresponding argument is an `int` bit-field. The formation of implicit conversion sequences treats the `int` bit-field as an `int` lvalue and finds an exact match with the parameter. If the function is selected by overload resolution, the call will nonetheless be ill-formed because of the prohibition on binding a non-`const` lvalue reference to a bit-field (8.2156.1). — *end example*]

13.903.0.0.1 List-initialization sequence [over.ics.list]

1

When an argument is an initializer list (8.2435.1), it is not an expression and special rules apply for converting it to a parameter type.

1

If the parameter type is `std::initializer_list<X>` and all the elements of the initializer list can be implicitly converted to `X`, the implicit conversion sequence is the worst conversion necessary to convert an element of the list to `X`, or if the initializer list has no elements, the identity conversion. This conversion can be a user-defined conversion even in the context of a call to an initializer-list constructor. [*Example:*

```
void f(std::initializer_list<int>);
```

```
f( {} );                    // OK: f(initializer_list<int>) identity conversion
f( {1,2,3} );               // OK: f(initializer_list<int>) identity conversion
f( {'a','b'} );             // OK: f(initializer_list<int>) integral promotion
f( {1.0} );                 // error: narrowing

struct A {
  A(std::initializer_list<double>);           // #1
  A(std::initializer_list<complex<double>>);  // #2
  A(std::initializer_list<std::string>);      // #3
};
A a{ 1.0,2.0 };             // OK, uses #1

void g(A);
g({ "foo", "bar" });        // OK, uses #3

typedef int IA[3];
void h(const IA&);
h({ 1, 2, 3 });             // OK: identity conversion
```

— *end example*]

1

Otherwise, if the parameter type is "array of `N X`"[135], if the initializer list has exactly `N` elements or if it has fewer than `N` elements and `X` is default-constructible, and if all the elements of the initializer list can be implicitly converted to `X`, the implicit conversion sequence is the worst conversion necessary to convert an element of the list to `X`.

1

Otherwise, if the parameter is a non-aggregate class `X` and overload resolution per 13.425.0.1 chooses a single best constructor of `X` to perform the initialization of an object of type `X` from the argument initializer list, the implicit conversion sequence is a user-defined conversion sequence with the second standard conversion sequence an identity conversion. If multiple constructors are viable but none is better than the others, the implicit conversion sequence is the ambiguous conversion sequence. User-defined conversions are allowed for conversion of the initializer list elements to the constructor parameter types except as noted in 13.779.0.1. [*Example:*

```
struct A {
  A(std::initializer_list<int>);
};
void f(A);
f( {'a', 'b'} );            // OK: f(A(std::initializer_list<int>)) user-defined conversion

struct B {
  B(int, double);
};
void g(B);
g( {'a', 'b'} );            // OK: g(B(int,double)) user-defined conversion
g( {1.0, 1.0} );            // error: narrowing

void f(B);
f( {'a', 'b'} );            // error: ambiguous f(A) or f(B)

struct C {
```

135) Since there are no parameters of array type, this will only occur as the underlying type of a reference parameter.

```
  C(std::string);
};
void h(C);
h({"foo"});                 // OK: h(C(std::string("foo")))

struct D {
  D(A, C);
};
void i(D);
i({ {1,2}, {"bar"} });      // OK: i(D(A(std::initializer_list<int>{1,2}),C(std::string("bar"))))
```

— *end example*]

1

Otherwise, if the parameter has an aggregate type which can be initialized from the initializer list according to the rules for aggregate initialization (8.1817.1), the implicit conversion sequence is a user-defined conversion sequence with the second standard conversion sequence an identity conversion. [*Example:*

```
struct A {
  int m1;
  double m2;
};

void f(A);
f( {'a', 'b'} );            // OK: f(A(int,double)) user-defined conversion
f( {1.0} );                 // error: narrowing
```

— *end example*]

1

Otherwise, if the parameter is a reference, see 13.870.0.0.1. [*Note:* The rules in this section will apply for initializing the underlying temporary for the reference. — *end note*] [*Example:*

```
struct A {
  int m1;
  double m2;
};

void f(const A&);
f( {'a', 'b'} );            // OK: f(A(int,double)) user-defined conversion
f( {1.0} );                 // error: narrowing

void g(const double &);
g({1});                     // same conversion as int to double
```

— *end example*]

1

Otherwise, if the parameter type is not a class:

— if the initializer list has one element, the implicit conversion sequence is the one required to convert the element to the parameter type; [*Example:*

```
void f(int);
f( {'a'} );                 // OK: same conversion as char to int
f( {1.0} );                 // error: narrowing
```

— *end example*]

— if the initializer list has no elements, the implicit conversion sequence is the identity conversion. [*Example:*

```
void f(int);
f( { } );                    // OK: identity conversion
```

— *end example*]

1

In all cases other than those enumerated above, no conversion is possible.

13.946.0.1 Ranking implicit conversion sequences [over.ics.rank]

1

13.946.0.1 defines a partial ordering of implicit conversion sequences based on the relationships *better conversion sequence* and *better conversion*. If an implicit conversion sequence S1 is defined by these rules to be a better conversion sequence than S2, then it is also the case that S2 is a *worse conversion sequence* than S1. If conversion sequence S1 is neither better than nor worse than conversion sequence S2, S1 and S2 are said to be *indistinguishable conversion sequences*.

1

When comparing the basic forms of implicit conversion sequences (as defined in 13.779.0.1)

— a standard conversion sequence (13.837.0.0.1) is a better conversion sequence than a user-defined conversion sequence or an ellipsis conversion sequence, and

— a user-defined conversion sequence (13.844.0.0.1) is a better conversion sequence than an ellipsis conversion sequence (13.868.0.0.1).

1

Two implicit conversion sequences of the same form are indistinguishable conversion sequences unless one of the following rules applies:

— Standard conversion sequence `S1` is a better conversion sequence than standard conversion sequence `S2` if

  — `S1` is a proper subsequence of `S2` (comparing the conversion sequences in the canonical form defined by 13.837.0.0.1, excluding any Lvalue Transformation; the identity conversion sequence is considered to be a subsequence of any non-identity conversion sequence) or, if not that,

  — the rank of `S1` is better than the rank of `S2`, or `S1` and `S2` have the same rank and are distinguishable by the rules in the paragraph below, or, if not that,

  — `S1` and `S2` are reference bindings (8.2156.1) and neither refers to an implicit object parameter of a non-static member function declared without a *ref-qualifier*, and `S1` binds an rvalue reference to an rvalue and `S2` binds an lvalue reference.

  [*Example:*

```
int i;
int f1();
int&& f2();
int g(const int&);
int g(const int&&);
int j = g(i);                    // calls g(const int&)
int k = g(f1());                 // calls g(const int&&)
int l = g(f2());                 // calls g(const int&&)
```

```
struct A {
  A& operator<<(int);
  void p() &;
  void p() &&;
};
A& operator<<(A&&, char);
A() << 1;                       // calls A::operator<<(int)
A() << 'c';                     // calls operator<<(A&&, char)
A a;
a << 1;                         // calls A::operator<<(int)
a << 'c';                       // calls A::operator<<(int)
A().p();                        // calls A::p()&&
a.p();                          // calls A::p()&
```

— *end example*] or, if not that,

— `S1` and `S2` are reference bindings (8.2156.1) and `S1` binds an lvalue reference to a function lvalue and `S2` binds an rvalue reference to a function lvalue. [*Example:*

```
int f(void(&)());               // #1
int f(void(&&)());              // #2
void g();
int i1 = f(g);                  // calls #1
```

— *end example*] or, if not that,

— `S1` and `S2` differ only in their qualification conversion and yield similar types `T1` and `T2` (4.246), respectively, and the cv-qualification signature of type `T1` is a proper subset of the cv-qualification signature of type `T2`. [*Example:*

```
int f(const volatile int *);
int f(const int *);
int i;
int j = f(&i);                  // calls f(const int*)
```

— *end example*] or, if not that,

— `S1` and `S2` are reference bindings (8.2156.1), and the types to which the references refer are the same type except for top-level cv-qualifiers, and the type to which the reference initialized by `S2` refers is more cv-qualified than the type to which the reference initialized by `S1` refers. [*Example:*

```
int f(const int &);
int f(int &);
int g(const int &);
int g(int);

int i;
int j = f(i);                   // calls f(int &)
int k = g(i);                   // ambiguous

struct X {
  void f() const;
  void f();
};
void g(const X& a, X b) {
  a.f();                        // calls X::f() const
  b.f();                        // calls X::f()
}
```

— *end example*]

— User-defined conversion sequence `U1` is a better conversion sequence than another user-defined conversion sequence `U2` if they contain the same user-defined conversion function or constructor or they initialize the same class in an aggregate initialization and in either case the second standard conversion sequence of `U1` is better than the second standard conversion sequence of `U2`. [*Example:*

```
struct A {
  operator short();
} a;
int f(int);
int f(float);
int i = f(a);                    // calls f(int), because short → int is
                                 // better than short → float.
```

— *end example*]

— List-initialization sequence `L1` is a better conversion sequence than list-initialization sequence `L2` if

  — `L1` converts to `std::initializer_list<X>` for some `X` and `L2` does not, or, if not that,

  — `L1` converts to type "array of `N1 T`", `L2` converts to type "array of `N2 T`", and `N1` is smaller than `N2`.

1

Standard conversion sequences are ordered by their ranks: an Exact Match is a better conversion than a Promotion, which is a better conversion than a Conversion. Two conversion sequences with the same rank are indistinguishable unless one of the following rules applies:

— A conversion that does not convert a pointer, a pointer to member, or `std::nullptr_t` to `bool` is better than one that does.

— A conversion that promotes an enumeration whose underlying type is fixed to its underlying type is better than one that promotes to the promoted underlying type, if the two are different.

— If class `B` is derived directly or indirectly from class `A`, conversion of `B*` to `A*` is better than conversion of `B*` to `void*`, and conversion of `A*` to `void*` is better than conversion of `B*` to `void*`.

— If class `B` is derived directly or indirectly from class `A` and class `C` is derived directly or indirectly from `B`,

  — conversion of `C*` to `B*` is better than conversion of `C*` to `A*`, [*Example:*

```
struct A {};
struct B : public A {};
struct C : public B {};
C* pc;
int f(A*);
int f(B*);
int i = f(pc);                   // calls f(B*)
```

  — *end example*]

  — binding of an expression of type `C` to a reference to type `B` is better than binding an expression of type `C` to a reference to type `A`,

  — conversion of `A::*` to `B::*` is better than conversion of `A::*` to `C::*`,

  — conversion of `C` to `B` is better than conversion of `C` to `A`,

  — conversion of `B*` to `A*` is better than conversion of `C*` to `A*`,

— binding of an expression of type `B` to a reference to type `A` is better than binding an expression of type `C` to a reference to type `A`,

— conversion of `B::*` to `C::*` is better than conversion of `A::*` to `C::*`, and

— conversion of `B` to `A` is better than conversion of `C` to `A`.

[*Note:* Compared conversion sequences will have different source types only in the context of comparing the second standard conversion sequence of an initialization by user-defined conversion (see 13.562.1); in all other contexts, the source types will be the same and the target types will be different. — *end note*]

13.1084 Address of overloaded function [over.over]

1

A use of an overloaded function name without arguments is resolved in certain contexts to a function, a pointer to function or a pointer to member function for a specific function from the overload set. A function template name is considered to name a set of overloaded functions in such contexts. The function selected is the one whose type is identical to the function type of the target type required in the context. [*Note:* That is, the class of which the function is a member is ignored when matching a pointer-to-member-function type. — *end note*] The target can be

— an object or reference being initialized (8.821, 8.2156.1, 8.2435.1),

— the left side of an assignment (5.3483),

— a parameter of a function (5.955.1),

— a parameter of a user-defined operator (13.1122),

— the return value of a function, operator function, or conversion (6.270.2),

— an explicit type conversion (5.1108.1, 5.1443.1, 5.2765), or

— a non-type *template-parameter* (14.364.1).

The overloaded function name can be preceded by the `&` operator. An overloaded function name shall not be used without arguments in contexts other than those listed. [*Note:* Any redundant set of parentheses surrounding the overloaded function name is ignored (5.398). — *end note*]

1

If the name is a function template, template argument deduction is done (14.2407.0.1), and if the argument deduction succeeds, the resulting template argument list is used to generate a single function template specialization, which is added to the set of overloaded functions considered. [*Note:* As described in 14.2075.1, if deduction fails and the function template name is followed by an explicit template argument list, the *template-id* is then examined to see whether it identifies a single function template specialization. If it does, the *template-id* is considered to be an lvalue for that function template specialization. The target type is not used in that determination. — *end note*]

1

Non-member functions and static member functions match targets of type "pointer-to-function" or "reference-to-function." Nonstatic member functions match targets of type "pointer-to-member-function". If a nonstatic member function is selected, the reference to the overloaded function name is required to have the form of a pointer to member as described in 5.1898.1.

1

If more than one function is selected, any function template specializations in the set are eliminated if the set also contains a function that is not a function template specialization, and any given function

template specialization F1 is eliminated if the set contains a second function template specialization whose function template is more specialized than the function template of F1 according to the partial ordering rules of 14.992.0.1. After such eliminations, if any, there shall remain exactly one selected function.

1 [*Example:*

```
int f(double);
int f(int);
int (*pfd)(double) = &f;          // selects f(double)
int (*pfi)(int) = &f;             // selects f(int)
int (*pfe)(...) = &f;             // error: type mismatch
int (&rfi)(int) = f;              // selects f(int)
int (&rfd)(double) = f;           // selects f(double)
void g() {
  (int (*)(int))&f;               // cast expression as selector
}
```

The initialization of pfe is ill-formed because no f() with type int(...) has been declared, and not because of any ambiguity. For another example,

```
struct X {
  int f(int);
  static int f(long);
};

int (X::*p1)(int)  = &X::f;      // OK
int    (*p2)(int)  = &X::f;      // error: mismatch
int    (*p3)(long) = &X::f;      // OK
int (X::*p4)(long) = &X::f;      // error: mismatch
int (X::*p5)(int)  = &(X::f);    // error: wrong syntax for
                                 // pointer to member
int    (*p6)(long) = &(X::f);    // OK
```

— *end example*]

2

[*Note:* If f() and g() are both overloaded functions, the cross product of possibilities must be considered to resolve f(&g), or the equivalent expression f(g). — *end note*]

1 [*Note:* There are no standard conversions (Clause 4) of one pointer-to-function type into another. In particular, even if B is a public base of D, we have

```
D* f();
B* (*p1)() = &f;                 // error

void g(D*);
void (*p2)(B*) = &g;             // error
```

— *end note*]

13.1122 Overloaded operators [over.oper]

1

A function declaration having one of the following *operator-function-id*s as its name declares an *operator function*. A function template declaration having one of the following *operator-function-id*s as its name declares an *operator function template*. A specialization of an operator function template is also an operator function. An operator function is said to *implement* the operator named in its *operator-function-id*.

operator-function-id:
`operator` *operator*

operator: one of

```
new   delete   new[]   delete[]
+     -        *       /        %        ^        &        |        ~
!     =        <       >        +=       -=       *=       /=       %=
^=    &=       |=      <<       >>       >>=      <<=      ==       !=
<=    >=       &&      ||       ++       --       ,        ->*      ->
()    []
```

[*Note:* The last two operators are function call (5.955.1) and subscripting (5.896.1). The operators `new[]`, `delete[]`, `()`, and `[]` are formed from more than one token. — *end note*]

1

Both the unary and binary forms of

```
+    -    *    &
```

can be overloaded.

1

The following operators cannot be overloaded:

```
.    .*    ::    ?:
```

nor can the preprocessing symbols `#` and `##` (Clause 16).

1

Operator functions are usually not called directly; instead they are invoked to evaluate the operators they implement (13.1166.1 – 13.1232.1). They can be explicitly called, however, using the *operator-function-id* as the name of the function in the function call syntax (5.955.1). [*Example:*

```
complex z = a.operator+(b);      // complex z = a+b;
void* p = operator new(sizeof(int)*n);
```

— *end example*]

1

The allocation and deallocation functions, `operator new`, `operator new[]`, `operator delete` and `operator delete[]`, are described completely in 3.2331.1. The attributes and restrictions found in the rest of this subclause do not apply to them unless explicitly stated in 3.2331.1.

1

An operator function shall either be a non-static member function or be a non-member function that has at least one parameter whose type is a class, a reference to a class, an enumeration, or a reference to an enumeration. It is not possible to change the precedence, grouping, or number of operands of operators. The meaning of the operators `=`, (unary) `&`, and `,` (comma), predefined for each type, can be changed for specific class and enumeration types by defining operator functions that implement these operators. Operator functions are inherited in the same manner as other base class functions.

1

The identities among certain predefined operators applied to basic types (for example, `++a` ≡ `a+=1`) need not hold for operator functions. Some predefined operators, such as `+=`, require an operand to be an lvalue when applied to basic types; this is not required by operator functions.

1

An operator function cannot have default arguments (8.545.1), except where explicitly stated below. Operator functions cannot have more or fewer parameters than the number required for the corresponding operator, as described in the rest of this subclause.

1

Operators not mentioned explicitly in subclauses 13.1178.1 through 13.1232.1 act as ordinary unary and binary operators obeying the rules of 13.1166.1 or 13.1170.1.

13.1166.1 Unary operators [over.unary]

1

A prefix unary operator shall be implemented by a non-static member function (9.401) with no parameters or a non-member function with one parameter. Thus, for any prefix unary operator `@`, `@x` can be interpreted as either `x.operator@()` or `operator@(x)`. If both forms of the operator function have been declared, the rules in 13.264.0.1 determine which, if any, interpretation is used. See 13.1232.1 for an explanation of the postfix unary operators `++` and `--`.

1 The unary and binary forms of the same operator are considered to have the same name. [*Note:* Consequently, a unary operator can hide a binary operator from an enclosing scope, and vice versa. — *end note*]

13.1170.1 Binary operators [over.binary]

1

A binary operator shall be implemented either by a non-static member function (9.401) with one parameter or by a non-member function with two parameters. Thus, for any binary operator `@`, `x@y` can be interpreted as either `x.operator@(y)` or `operator@(x,y)`. If both forms of the operator function have been declared, the rules in 13.264.0.1 determine which, if any, interpretation is used.

13.1178.1 Assignment [over.ass]

1

An assignment operator shall be implemented by a non-static member function with exactly one parameter. Because a copy assignment operator `operator=` is implicitly declared for a class if not declared by the user (12.1577), a base class assignment operator is always hidden by the copy assignment operator of the derived class.

1

Any assignment operator, even the copy and move assignment operators, can be virtual. [*Note:* For a derived class `D` with a base class `B` for which a virtual copy/move assignment has been declared, the copy/move assignment operator in `D` does not override `B`'s virtual copy/move assignment operator. [*Example:*

```
struct B {
  virtual int operator= (int);
  virtual B& operator= (const B&);
};
struct D : B {
  virtual int operator= (int);
  virtual D& operator= (const B&);
};

D dobj1;
D dobj2;
B* bptr = &dobj1;
void f() {
```

```
  bptr->operator=(99);          // calls D::operator=(int)
  *bptr = 99;                   // ditto
  bptr->operator=(dobj2);       // calls D::operator=(const B&)
  *bptr = dobj2;                // ditto
  dobj1 = dobj2;                // calls implicitly-declared
                                // D::operator=(const D&)
}
```

— *end example*] — *end note*]

13.1196.1 Function call [over.call]

1

`operator()` shall be a non-static member function with an arbitrary number of parameters. It can have default arguments. It implements the function call syntax

postfix-expression `(` *expression-list*$_{opt}$`)`

where the *postfix-expression* evaluates to a class object and the possibly empty *expression-list* matches the parameter list of an `operator()` member function of the class. Thus, a call `x(arg1,...)` is interpreted as `x.operator()(arg1, ...)` for a class object `x` of type `T` if `T::operator()(T1, T2, T3)` exists and if the operator is selected as the best match function by the overload resolution mechanism (13.562.1).

13.1210.1 Subscripting [over.sub]

1

`operator[]` shall be a non-static member function with exactly one parameter. It implements the subscripting syntax

postfix-expression `[` *expression* `]`

or

postfix-expression `[` *braced-init-list* `]`

Thus, a subscripting expression `x[y]` is interpreted as `x.operator[](y)` for a class object `x` of type `T` if `T::operator[](T1)` exists and if the operator is selected as the best match function by the overload resolution mechanism (13.562.1). [*Example:*

```
struct X {
  Z operator[](std::initializer_list<int>);
};
X x;
x[{1,2,3}] = 7;          // OK: meaning x.operator[]({1,2,3})
int a[10];
a[{1,2,3}] = 7;          // error: built-in subscript operator
```

— *end example*]

13.1217.1 Class member access [over.ref]

1

`operator->` shall be a non-static member function taking no parameters. It implements the class member access syntax that uses `->`.

postfix-expression `->` `template`$_{opt}$ *id-expression*
postfix-expression `->` *pseudo-destructor-name*

An expression `x->m` is interpreted as `(x.operator->())->m` for a class object `x` of type `T` if `T::operator->()` exists and if the operator is selected as the best match function by the overload resolution mechanism (13.88).

13.1232.1 Increment and decrement [over.inc]

1

The user-defined function called `operator++` implements the prefix and postfix `++` operator. If this function is a non-static member function with no parameters, or a non-member function with one parameter, it defines the prefix increment operator `++` for objects of that type. If the function is a non-static member function with one parameter (which shall be of type `int`) or a non-member function with two parameters (the second of which shall be of type `int`), it defines the postfix increment operator `++` for objects of that type. When the postfix increment is called as a result of using the `++` operator, the `int` argument will have value zero.[136] [*Example:*

```
struct X {
  X&   operator++();            // prefix ++a
  X    operator++(int);         // postfix a++
};

struct Y { };
Y&   operator++(Y&);            // prefix ++b
Y    operator++(Y&, int);       // postfix b++

void f(X a, Y b) {
  ++a;                          // a.operator++();
  a++;                          // a.operator++(0);
  ++b;                          // operator++(b);
  b++;                          // operator++(b, 0);

  a.operator++();               // explicit call: like ++a;
  a.operator++(0);              // explicit call: like a++;
  operator++(b);                // explicit call: like ++b;
  operator++(b, 0);             // explicit call: like b++;
}
```

— *end example*]

1 The prefix and postfix decrement operators `--` are handled analogously.

13.1241.1 User-defined literals [over.literal]

literal-operator-id:
`operator` *string-literal identifier*
`operator` *user-defined-string-literal*

1 The *string-literal* or *user-defined-string-literal* in a *literal-operator-id* shall have no *encoding-prefix* and shall contain no characters other than the implicit terminating `'\0'`. The *ud-suffix* of the *user-defined-string-literal* or the *identifier* in a *literal-operator-id* is called a *literal suffix identifier*. [*Note:* some literal suffix identifiers are reserved for future standardization; see 17.424.0.0.1. — *end note*]

2

A declaration whose *declarator-id* is a *literal-operator-id* shall be a declaration of a namespace-scope function or function template (it could be a friend function (11.94)), an explicit instantiation or specialization of a function template, or a *using-declaration* (7.1311.1). A function declared with a *literal-operator-id* is a *literal operator*. A function template declared with a *literal-operator-id* is a *literal operator template*.

1

136) Calling `operator++` explicitly, as in expressions like `a.operator++(2)`, has no special properties: The argument to `operator++` is 2.

The declaration of a literal operator shall have a *parameter-declaration-clause* equivalent to one of the following:

```
const char*
unsigned long long int
long double
char
wchar_t
char16_t
char32_t
const char*, std::size_t
const wchar_t*, std::size_t
const char16_t*, std::size_t
const char32_t*, std::size_t
```

If a parameter has a default argument (8.545.1), the program is ill-formed.

1 A *raw literal operator* is a literal operator with a single parameter whose type is `const char*`.

2

The declaration of a literal operator template shall have an empty *parameter-declaration-clause* and its *template-parameter-list* shall have a single *template-parameter* that is a non-type template parameter pack (14.703.1) with element type `char`.

1 Literal operators and literal operator templates shall not have C language linkage.

2

[*Note:* Literal operators and literal operator templates are usually invoked implicitly through user-defined literals (2.622.1). However, except for the constraints described above, they are ordinary namespace-scope functions and function templates. In particular, they are looked up like ordinary functions and function templates and they follow the same overload resolution rules. Also, they can be declared `inline` or `constexpr`, they may have internal or external linkage, they can be called explicitly, their addresses can be taken, etc. — *end note*]

1 [*Example:*

```
void operator "" _km(long double);                  // OK
string operator "" _i18n(const char*, std::size_t); // OK
template <char...> double operator "" _\u03C0();    // OK: UCN for lowercase pi
float operator ""_e(const char*);                   // OK
float operator ""E(const char*);                    // error: reserved literal suffix (17.424.0.0.1, 2.622.1)
double operator""_Bq(long double);                  // OK: does not use the reserved name _Bq (17.332.0.0.1)
double operator"" _Bq(long double);                 // uses the reserved name _Bq (17.332.0.0.1)
float operator " " B(const char*);                  // error: non-empty string-literal
string operator "" 5X(const char*, std::size_t);    // error: invalid literal suffix identifier
double operator "" _miles(double);                  // error: invalid parameter-declaration-clause
template <char...> int operator "" _j(const char*); // error: invalid parameter-declaration-clause
extern "C" void operator "" _m(long double);        // error: C language linkage
```

— *end example*]

13.1258 Built-in operators [over.built]

1

The candidate operator functions that represent the built-in operators defined in Clause 5 are specified in this subclause. These candidate functions participate in the operator overload resolution process as described in 13.264.0.1 and are used for no other purpose. [*Note:* Because built-in operators take only operands with

non-class type, and operator overload resolution occurs only when an operand expression originally has class or enumeration type, operator overload resolution can resolve to a built-in operator only when an operand has a class type that has a user-defined conversion to a non-class type appropriate for the operator, or when an operand has an enumeration type that can be converted to a type appropriate for the operator. Also note that some of the candidate operator functions given in this subclause are more permissive than the built-in operators themselves. As described in 13.264.0.1, after a built-in operator is selected by overload resolution the expression is subject to the requirements for the built-in operator given in Clause 5, and therefore to any additional semantic constraints given there. If there is a user-written candidate with the same name and parameter types as a built-in candidate operator function, the built-in operator function is hidden and is not included in the set of candidate functions. — *end note*]

1

In this subclause, the term *promoted integral type* is used to refer to those integral types which are preserved by integral promotion (including e.g. `int` and `long` but excluding e.g. `char`). Similarly, the term *promoted arithmetic type* refers to floating types plus promoted integral types. [*Note:* In all cases where a promoted integral type or promoted arithmetic type is required, an operand of enumeration type will be acceptable by way of the integral promotions. — *end note*]

1

For every pair (T, VQ), where T is an arithmetic type, and VQ is either `volatile` or empty, there exist candidate operator functions of the form

```
VQ T& operator++(VQ T&);
T operator++(VQ T&, int);
```

1 For every pair (T, VQ), where T is an arithmetic type other than *bool*, and VQ is either `volatile` or empty, there exist candidate operator functions of the form

```
VQ T& operator--(VQ T&);
T operator--(VQ T&, int);
```

2

For every pair (T, VQ), where T is a cv-qualified or cv-unqualified object type, and VQ is either `volatile` or empty, there exist candidate operator functions of the form

```
T*VQ& operator++(T*VQ&);
T*VQ& operator--(T*VQ&);
T*    operator++(T*VQ&, int);
T*    operator--(T*VQ&, int);
```

1 For every cv-qualified or cv-unqualified object type T, there exist candidate operator functions of the form

```
T&    operator*(T*);
```

2 For every function type T that does not have cv-qualifiers or a *ref-qualifier*, there exist candidate operator functions of the form

```
T&    operator*(T*);
```

3 For every type T there exist candidate operator functions of the form

```
T*    operator+(T*);
```

4 For every promoted arithmetic type T, there exist candidate operator functions of the form

```
T operator+(T);
T operator-(T);
```

5 For every promoted integral type *T*, there exist candidate operator functions of the form

```
T operator~(T);
```

6 For every quintuple (*C1*, *C2*, *T*, *CV1*, *CV2*), where *C2* is a class type, *C1* is the same type as C2 or is a derived class of C2, *T* is an object type or a function type, and *CV1* and *CV2* are *cv-qualifier-seq*s, there exist candidate operator functions of the form

```
CV12 T& operator->*(CV1 C1*, CV2 T C2::*);
```

where *CV12* is the union of *CV1* and *CV2*. The return type is shown for exposition only; see 5.2832 for the determination of the operator's result type.

7

For every pair of promoted arithmetic types *L* and *R*, there exist candidate operator functions of the form

```
LR      operator*(L, R);
LR      operator/(L, R);
LR      operator+(L, R);
LR      operator-(L, R);
bool    operator<(L, R);
bool    operator>(L, R);
bool    operator<=(L, R);
bool    operator>=(L, R);
bool    operator==(L, R);
bool    operator!=(L, R);
```

where *LR* is the result of the usual arithmetic conversions between types *L* and *R*.

1 For every cv-qualified or cv-unqualified object type *T* there exist candidate operator functions of the form

```
T*      operator+(T*, std::ptrdiff_t);
T&      operator[](T*, std::ptrdiff_t);
T*      operator-(T*, std::ptrdiff_t);
T*      operator+(std::ptrdiff_t, T*);
T&      operator[](std::ptrdiff_t, T*);
```

2 For every *T*, where *T* is a pointer to object type, there exist candidate operator functions of the form

```
std::ptrdiff_t   operator-(T, T);
```

3 For every *T*, where *T* is an enumeration type or a pointer type, there exist candidate operator functions of the form

```
bool    operator<(T, T);
bool    operator>(T, T);
bool    operator<=(T, T);
bool    operator>=(T, T);
bool    operator==(T, T);
bool    operator!=(T, T);
```

4 For every pointer to member type *T* or type `std::nullptr_t` there exist candidate operator functions of the form

```
bool    operator==(T, T);
bool    operator!=(T, T);
```

5

For every pair of promoted integral types *L* and *R*, there exist candidate operator functions of the form

```
LR      operator%(L, R);
LR      operator&(L, R);
LR      operator^(L, R);
LR      operator|(L, R);
L       operator<<(L, R);
L       operator>>(L, R);
```

where LR is the result of the usual arithmetic conversions between types L and R.

1 For every triple (L, VQ, R), where L is an arithmetic type, VQ is either `volatile` or empty, and R is a promoted arithmetic type, there exist candidate operator functions of the form

```
VQ L&   operator=(VQ L&, R);
VQ L&   operator*=(VQ L&, R);
VQ L&   operator/=(VQ L&, R);
VQ L&   operator+=(VQ L&, R);
VQ L&   operator-=(VQ L&, R);
```

2 For every pair (T, VQ), where T is any type and VQ is either `volatile` or empty, there exist candidate operator functions of the form

```
T*VQ&   operator=(T*VQ&, T*);
```

3 For every pair (T, VQ), where T is an enumeration or pointer to member type and VQ is either `volatile` or empty, there exist candidate operator functions of the form

```
VQ T&   operator=(VQ T&, T);
```

4 For every pair (T, VQ), where T is a cv-qualified or cv-unqualified object type and VQ is either `volatile` or empty, there exist candidate operator functions of the form

```
T*VQ&   operator+=(T*VQ&, std::ptrdiff_t);
T*VQ&   operator-=(T*VQ&, std::ptrdiff_t);
```

5 For every triple (L, VQ, R), where L is an integral type, VQ is either `volatile` or empty, and R is a promoted integral type, there exist candidate operator functions of the form

```
VQ L&   operator%=(VQ L&, R);
VQ L&   operator<<=(VQ L&, R);
VQ L&   operator>>=(VQ L&, R);
VQ L&   operator&=(VQ L&, R);
VQ L&   operator^=(VQ L&, R);
VQ L&   operator|=(VQ L&, R);
```

6

There also exist candidate operator functions of the form

```
bool    operator!(bool);
bool    operator&&(bool, bool);
bool    operator||(bool, bool);
```

1

For every pair of promoted arithmetic types L and R, there exist candidate operator functions of the form

```
LR      operator?:(bool, L, R);
```

where LR is the result of the usual arithmetic conversions between types L and R. [*Note:* As with all these descriptions of candidate functions, this declaration serves only to describe the built-in operator for purposes of overload resolution. The operator "`?:`" cannot be overloaded. — *end note*]

1

For every type T, where T is a pointer, pointer-to-member, or scoped enumeration type, there exist candidate operator functions of the form

```
T       operator?:(bool, T, T);
```

14 Templates [temp]

1

A *template* defines a family of classes or functions or an alias for a family of types.

template-declaration:
`template` < *template-parameter-list* > *declaration*

template-parameter-list:
template-parameter
template-parameter-list , *template-parameter*

[*Note:* The > token following the *template-parameter-list* of a *template-declaration* may be the product of replacing a >> token by two consecutive > tokens (14.224). — *end note*]

The *declaration* in a *template-declaration* shall

— declare or define a function, a class, or a variable, or

— define a member function, a member class, a member enumeration, or a static data member of a class template or of a class nested within a class template, or

— define a member template of a class or class template, or

— be an *alias-declaration*.

A *template-declaration* is a *declaration*. A *template-declaration* is also a definition if its *declaration* defines a function, a class, a variable, or a static data member. A declaration introduced by a template declaration of a variable is a *variable template*. A variable template at class scope is a *static data member template*.

[*Example:*

```
template<class T>
  constexpr T pi = T(3.1415926535897932385L);
template<class T>
T circular_area(T r) {
  return pi<T> * r * r;
}
struct matrix_constants {
  template<class T>
   using pauli = hermitian_matrix<T, 2>;
  template<class T>
   constexpr pauli<T> sigma1 = { { 0, 1 }, { 1, 0 } };
  template<class T>
   constexpr pauli<T> sigma2 = { { 0, -1i }, { 1i, 0 } };
  template<class T>
   constexpr pauli<T> sigma3 = { { 1, 0 }, { 0, -1 } };
};
```

— *end example*]

1

A *template-declaration* can appear only as a namespace scope or class scope declaration. In a function template declaration, the last component of the *declarator-id* shall not be a *template-id*. [*Note:* That last component may be an *identifier*, an *operator-function-id*, a *conversion-function-id*, or a *literal-operator-id*.

In a class template declaration, if the class name is a *simple-template-id*, the declaration declares a class template partial specialization (14.812.1). — *end note*]

1

In a *template-declaration*, explicit specialization, or explicit instantiation the *init-declarator-list* in the declaration shall contain at most one declarator. When such a declaration is used to declare a class template, no declarator is permitted.

1

A template name has linkage (3.1459). A non-member function template can have internal linkage; any other template name shall have external linkage. Specializations (explicit or implicit) of a template that has internal linkage are distinct from all specializations in other translation units. A template, a template explicit specialization (14.1921.1), and a class template partial specialization shall not have C linkage. Use of a linkage specification other than C or C++ with any of these constructs is conditionally-supported, with implementation-defined semantics. Template definitions shall obey the one definition rule (3.91). [*Note:* Default arguments for function templates and for member functions of class templates are considered definitions for the purpose of template instantiation (14.616) and must also obey the one definition rule. — *end note*]

1

A class template shall not have the same name as any other template, class, function, variable, enumeration, enumerator, namespace, or type in the same scope (3.464), except as specified in (14.812.1). Except that a function template can be overloaded either by non-template functions (8.360.1) with the same name or by other function templates with the same name (14.2662.1), a template name declared in namespace scope or in class scope shall be unique in that scope.

1

A function template, member function of a class template, variable template, or static data member of a class template shall be defined in every translation unit in which it is implicitly instantiated (14.1644.1) unless the corresponding specialization is explicitly instantiated (14.1841.1) in some translation unit; no diagnostic is required.

14.97 Template parameters [temp.param]

1 The syntax for *template-parameter*s is:

template-parameter:
  type-parameter
  parameter-declaration

type-parameter:
  `class` `...`$_{opt}$ *identifier*$_{opt}$
  `class` *identifier*$_{opt}$`=` *type-id*
  `typename` `...`$_{opt}$ *identifier*$_{opt}$
  `typename` *identifier*$_{opt}$`=` *type-id*
  `template` `<` *template-parameter-list* `>` `class` `...`$_{opt}$ *identifier*$_{opt}$
  `template` `<` *template-parameter-list* `>` `class` *identifier*$_{opt}$`=` *id-expression*

[*Note:* The `>` token following the *template-parameter-list* of a *type-parameter* may be the product of replacing a `>>` token by two consecutive `>` tokens (14.224). — *end note*]

2

There is no semantic difference between `class` and `typename` in a *template-parameter*. `typename` followed by an *unqualified-id* names a template type parameter. `typename` followed by a *qualified-id* denotes the type in a non-type [137] *parameter-declaration*. A storage class shall not be specified in a *template-parameter*

137) Since template *template-parameter*s and template *template-argument*s are treated as types for descriptive purposes, the

declaration. Types shall not be defined in a *template-parameter* declaration. [*Note:* A template parameter may be a class template. For example,

```
template<class T> class myarray { /* ... */ };

template<class K, class V, template<class T> class C = myarray>
class Map {
  C<K> key;
  C<V> value;
};
```

— *end note*]

1

A *type-parameter* whose identifier does not follow an ellipsis defines its *identifier* to be a *typedef-name* (if declared with `class` or `typename`) or *template-name* (if declared with `template`) in the scope of the template declaration. [*Note:* Because of the name lookup rules, a *template-parameter* that could be interpreted as either a non-type *template-parameter* or a *type-parameter* (because its *identifier* is the name of an already existing class) is taken as a *type-parameter*. For example,

```
class T { /* ... */ };
int i;

template<class T, T i> void f(T t) {
  T t1 = i;          // template-parameters T and i
  ::T t2 = ::i;      // global namespace members T and i
}
```

Here, the template `f` has a *type-parameter* called `T`, rather than an unnamed non-type *template-parameter* of class `T`. — *end note*]

1

A non-type *template-parameter* shall have one of the following (optionally *cv-qualified*) types:

— integral or enumeration type,

— pointer to object or pointer to function,

— lvalue reference to object or lvalue reference to function,

— pointer to member,

— `std::nullptr_t`.

1

[*Note:* Other types are disallowed either explicitly below or implicitly by the rules governing the form of *template-arguments* (14.326). — *end note*] The top-level *cv-qualifiers* on the *template-parameter* are ignored when determining its type.

1

A non-type non-reference *template-parameter* is a prvalue. It shall not be assigned to or in any other way have its value changed. A non-type non-reference *template-parameter* cannot have its address taken. When a non-type non-reference *template-parameter* is used as an initializer for a reference, a temporary is always used. [*Example:*

terms *non-type parameter* and *non-type argument* are used to refer to non-type, non-template parameters and arguments.

```
template<const X& x, int i> void f() {
  i++;                          // error: change of template-parameter value

  &x;                           // OK
  &i;                           // error: address of non-reference template-parameter

  int& ri = i;                  // error: non-const reference bound to temporary
  const int& cri = i;           // OK: const reference bound to temporary
}
```

— *end example*]

1

A non-type *template-parameter* shall not be declared to have floating point, class, or void type. [*Example:*

```
template<double d> class X;     // error
template<double* pd> class Y;   // OK
template<double& rd> class Z;   // OK
```

— *end example*]

1

A non-type *template-parameter* of type "array of `T`" or "function returning `T`" is adjusted to be of type "pointer to `T`" or "pointer to function returning `T`", respectively. [*Example:*

```
template<int* a>   struct R { /* ... */ };
template<int b[5]> struct S { /* ... */ };
int p;
R<&p> w;                        // OK
S<&p> x;                        // OK due to parameter adjustment
int v[5];
R<v> y;                         // OK due to implicit argument conversion
S<v> z;                         // OK due to both adjustment and conversion
```

— *end example*]

1

A *default template-argument* is a *template-argument* (14.326) specified after `=` in a *template-parameter*. A default *template-argument* may be specified for any kind of *template-parameter* (type, non-type, template) that is not a template parameter pack (14.703.1). A default *template-argument* may be specified in a template declaration. A default *template-argument* shall not be specified in the *template-parameter-lists* of the definition of a member of a class template that appears outside of the member's class. A default *template-argument* shall not be specified in a friend class template declaration. If a friend function template declaration specifies a default *template-argument*, that declaration shall be a definition and shall be the only declaration of the function template in the translation unit.

1

The set of default *template-arguments* available for use with a template declaration or definition is obtained by merging the default arguments from the definition (if in scope) and all declarations in scope in the same way default function arguments are (8.545.1). [*Example:*

```
template<class T1, class T2 = int> class A;
template<class T1 = int, class T2> class A;
```

is equivalent to

```
template<class T1 = int, class T2 = int> class A;
```

— *end example*]

1

If a *template-parameter* of a class template or alias template has a default *template-argument*, each subsequent *template-parameter* shall either have a default *template-argument* supplied or be a template parameter pack. If a *template-parameter* of a primary class template or alias template is a template parameter pack, it shall be the last *template-parameter*. A template parameter pack of a function template shall not be followed by another template parameter unless that template parameter can be deduced from the *parameter-type-list* of the function template or has a default argument (14.2114.1). [*Example:*

```
template<class T1 = int, class T2> class B;   // error

// U can be neither deduced from the parameter-type-list nor specified
template<class... T, class... U> void f() { } // error
template<class... T, class U> void g() { }    // error
```

— *end example*]

1

A *template-parameter* shall not be given default arguments by two different declarations in the same scope. [*Example:*

```
template<class T = int> class X;
template<class T = int> class X { /*... */ }; // error
```

— *end example*]

1 When parsing a default *template-argument* for a non-type *template-parameter*, the first non-nested `>` is taken as the end of the *template-parameter-list* rather than a greater-than operator. [*Example:*

```
template<int i = 3 > 4 >        // syntax error
class X { /* ... */ };

template<int i = (3 > 4) >      // OK
class Y { /* ... */ };
```

— *end example*]

2

A *template-parameter* of a template *template-parameter* is permitted to have a default *template-argument*. When such default arguments are specified, they apply to the template *template-parameter* in the scope of the template *template-parameter*. [*Example:*

```
template <class T = float> struct B {};
template <template <class TT = float> class T> struct A {
  inline void f();
  inline void g();
};
template <template <class TT> class T> void A<T>::f() {
  T<> t;            // error - TT has no default template argument
}
template <template <class TT = char> class T> void A<T>::g() {
    T<> t;          // OK - T<char>
}
```

— *end example*]

1

If a *template-parameter* is a *type-parameter* with an ellipsis prior to its optional *identifier* or is a *parameter-declaration* that declares a parameter pack (8.360.1), then the *template-parameter* is a template parameter pack (14.703.1). A template parameter pack that is a *parameter-declaration* whose type contains one or more unexpanded parameter packs is a pack expansion. Similarly, a template parameter pack that is a *type-parameter* with a *template-parameter-list* containing one or more unexpanded parameter packs is a pack expansion. A template parameter pack that is a pack expansion shall not expand a parameter pack declared in the same *template-parameter-list*. [*Example:*

```
template <class... Types> class Tuple;                  // Types is a template type parameter pack
                                                        // but not a pack expansion
template <class T, int... Dims> struct multi_array;     // Dims is a non-type template parameter pack
                                                        // but not a pack expansion
template<class... T> struct value_holder {
  template<T... Values> struct apply { };               // Values is a non-type template parameter pack
                                                        // and a pack expansion
};
template<class... T, T... Values> struct static_array;// error: Values expands template type parameter
                                                        // pack T within the same template parameter list
```

— *end example*]

14.224 Names of template specializations [temp.names]

1

A template specialization (14.1614) can be referred to by a *template-id*:

simple-template-id:
  template-name < *template-argument-list*$_{opt}$ >

template-id:
  simple-template-id
  operator-function-id < *template-argument-list*$_{opt}$ >
  literal-operator-id < *template-argument-list*$_{opt}$ >

template-name:
  identifier

template-argument-list:
  template-argument ...$_{opt}$
  template-argument-list , *template-argument* ...$_{opt}$

template-argument:
  constant-expression
  type-id
  id-expression

[*Note:* The name lookup rules (3.923) are used to associate the use of a name with a template declaration; that is, to identify a name as a *template-name*. — *end note*]

1

For a *template-name* to be explicitly qualified by the template arguments, the name must be known to refer to a template.

1

After name lookup (3.923) finds that a name is a *template-name* or that an *operator-function-id* or a *literal-operator-id* refers to a set of overloaded functions any member of which is a function template, if this is followed by a <, the < is always taken as the delimiter of a *template-argument-list* and never as the less-than operator. When parsing a *template-argument-list*, the first non-nested >[138] is taken as the ending delimiter

138) A > that encloses the *type-id* of a `dynamic_cast`, `static_cast`, `reinterpret_cast` or `const_cast`, or which encloses the

rather than a greater-than operator. Similarly, the first non-nested `>>` is treated as two consecutive but distinct `>` tokens, the first of which is taken as the end of the *template-argument-list* and completes the *template-id*. [*Note:* The second `>` token produced by this replacement rule may terminate an enclosing *template-id* construct or it may be part of a different construct (e.g. a cast). — *end note*] [*Example:*

```
template<int i> class X { /* ...  */ };

X< 1>2 > x1;                        // syntax error
X<(1>2)> x2;                        // OK

template<class T> class Y { /* ...  */ };
Y<X<1>> x3;                         // OK, same as Y<X<1> > x3;
Y<X<6>>1>> x4;                      // syntax error
Y<X<(6>>1)>> x5;                    // OK
```

— *end example*]

1

When the name of a member template specialization appears after `.` or `->` in a *postfix-expression* or after a *nested-name-specifier* in a *qualified-id*, and the object expression of the *postfix-expression* is type-dependent or the *nested-name-specifier* in the *qualified-id* refers to a dependent type, but the name is not a member of the current instantiation (14.1319.0.1), the member template name must be prefixed by the keyword `template`. Otherwise the name is assumed to name a non-template. [*Example:*

```
struct X {
  template<std::size_t> X* alloc();
  template<std::size_t> static X* adjust();
};
template<class T> void f(T* p) {
  T* p1 = p->alloc<200>();          // ill-formed: < means less than
  T* p2 = p->template alloc<200>(); // OK: < starts template argument list
  T::adjust<100>();                 // ill-formed: < means less than
  T::template adjust<100>();        // OK: < starts template argument list
}
```

— *end example*]

1

A name prefixed by the keyword `template` shall be a *template-id* or the name shall refer to a class template. [*Note:* The keyword `template` may not be applied to non-template members of class templates. — *end note*] [*Note:* As is the case with the `typename` prefix, the `template` prefix is allowed in cases where it is not strictly necessary; i.e., when the *nested-name-specifier* or the expression on the left of the `->` or `.` is not dependent on a *template-parameter*, or the use does not appear in the scope of a template. — *end note*] [*Example:*

```
template <class T> struct A {
  void f(int);
  template <class U> void f(U);
};

template <class T> void f(T t) {
  A<T> a;
  a.template f<>(t);                // OK: calls template
  a.template f(t);                  // error: not a template-id
```

template-arguments of a subsequent *template-id*, is considered nested for the purpose of this description.

```
}

template <class T> struct B {
  template <class T2> struct C { };
};

// OK: T::template C names a class template:
template <class T, template <class X> class TT = T::template C> struct D { };
D<B<int> > db;
```

— *end example*]

1

A *simple-template-id* that names a class template specialization is a *class-name* (Clause 9).

1 A *template-id* that names an alias template specialization is a *type-name*.

14.326 Template arguments [temp.arg]

1

There are three forms of *template-argument*, corresponding to the three forms of *template-parameter*: type, non-type and template. The type and form of each *template-argument* specified in a *template-id* shall match the type and form specified for the corresponding parameter declared by the template in its *template-parameter-list*. When the parameter declared by the template is a template parameter pack (14.703.1), it will correspond to zero or more *template-arguments*. [*Example:*

```
template<class T> class Array {
  T* v;
  int sz;
public:
  explicit Array(int);
  T& operator[](int);
  T& elem(int i) { return v[i]; }
};

Array<int> v1(20);
typedef std::complex<double> dcomplex;  // std::complex is a standard
                                        // library template
Array<dcomplex> v2(30);
Array<dcomplex> v3(40);

void bar() {
  v1[3] = 7;
  v2[3] = v3.elem(4) = dcomplex(7,8);
}
```

— *end example*]

1

In a *template-argument*, an ambiguity between a *type-id* and an expression is resolved to a *type-id*, regardless of the form of the corresponding *template-parameter*.[139] [*Example:*

```
template<class T> void f();
template<int I> void f();
```

139) There is no such ambiguity in a default *template-argument* because the form of the *template-parameter* determines the allowable forms of the *template-argument*.

```
void g() {
  f<int()>();        // int() is a type-id: call the first f()
}
```

— *end example*]

1

The name of a *template-argument* shall be accessible at the point where it is used as a *template-argument*. [*Note:* If the name of the *template-argument* is accessible at the point where it is used as a *template-argument*, there is no further access restriction in the resulting instantiation where the corresponding *template-parameter* name is used. — *end note*] [*Example:*

```
template<class T> class X {
  static T t;
};

class Y {
private:
  struct S { /* ... */ };
  X<S> x;            // OK: S is accessible
                     // X<Y::S> has a static member of type Y::S
                     // OK: even though Y::S is private
};

X<Y::S> y;           // error: S not accessible
```

— *end example*] For a *template-argument* that is a class type or a class template, the template definition has no special access rights to the members of the *template-argument*. [*Example:*

```
template <template <class TT> class T> class A {
  typename T<int>::S s;
};

template <class U> class B {
private:
  struct S { /* ... */ };
};

A<B> b;              // ill-formed: A has no access to B::S
```

— *end example*]

1

When template argument packs or default *template-argument*s are used, a *template-argument* list can be empty. In that case the empty <> brackets shall still be used as the *template-argument-list*. [*Example:*

```
template<class T = char> class String;
String<>* p;                    // OK: String<char>
String* q;                      // syntax error
template<class ... Elements> class Tuple;
Tuple<>* t;                     // OK: Elements is empty
Tuple* u;                       // syntax error
```

— *end example*]

1

An explicit destructor call (12.807) for an object that has a type that is a class template specialization may explicitly specify the *template-arguments*. [*Example:*

```
template<class T> struct A {
  ~A();
};
void f(A<int>* p, A<int>* q) {
  p->A<int>::~A();              // OK: destructor call
  q->A<int>::~A<int>();         // OK: destructor call
}
```

— *end example*]

1

If the use of a *template-argument* gives rise to an ill-formed construct in the instantiation of a template specialization, the program is ill-formed.

1 When the template in a *template-id* is an overloaded function template, both non-template functions in the overload set and function templates in the overload set for which the *template-arguments* do not match the *template-parameters* are ignored. If none of the function templates have matching *template-parameters*, the program is ill-formed.

2

A *template-argument* followed by an ellipsis is a pack expansion (14.703.1).

14.349.1 Template type arguments [temp.arg.type]

1

A *template-argument* for a *template-parameter* which is a type shall be a *type-id*.

1

[*Example:*

```
template <class T> class X { };
template <class T> void f(T t) { }
struct { } unnamed_obj;

void f() {
  struct A { };
  enum { e1 };
  typedef struct { } B;
  B b;
  X<A> x1;          // OK
  X<A*> x2;         // OK
  X<B> x3;          // OK
  f(e1);            // OK
  f(unnamed_obj);   // OK
  f(b);             // OK
}
```

— *end example*] [*Note:* A template type argument may be an incomplete type (3.2955). — *end note*]

1

If a declaration acquires a function type through a type dependent on a *template-parameter* and this causes a declaration that does not use the syntactic form of a function declarator to have function type, the program is ill-formed. [*Example:*

```
template<class T> struct A {
  static T t;
};
typedef int function();
A<function> a;                    // ill-formed: would declare A<function>::t
                                  // as a static member function
```

— *end example*]

14.364.1 Template non-type arguments [temp.arg.nontype]

1

A *template-argument* for a non-type, non-template *template-parameter* shall be one of:

— for a non-type *template-parameter* of integral or enumeration type, a converted constant expression (5.3643) of the type of the *template-parameter*; or

— the name of a non-type *template-parameter*; or

— a constant expression (5.3643) that designates the address of a complete object with static storage duration and external or internal linkage or a function with external or internal linkage, including function templates and function *template-id*s but excluding non-static class members, expressed (ignoring parentheses) as `&` *id-expression*, where the *id-expression* is the name of an object or function, except that the `&` may be omitted if the name refers to a function or array and shall be omitted if the corresponding *template-parameter* is a reference; or

— a constant expression that evaluates to a null pointer value (4.579); or

— a constant expression that evaluates to a null member pointer value (4.710); or

— a pointer to member expressed as described in 5.1898.1; or

— a constant expression of type `std::nullptr_t`.

1

[*Note:* A string literal (2.453.1) does not satisfy the requirements of any of these categories and thus is not an acceptable *template-argument*. [*Example:*

```
template<class T, const char* p> class X {
  /* ... */
};

X<int, "Studebaker"> x1;          // error: string literal as template-argument

const char p[] = "Vivisectionist";
X<int,p> x2;                      // OK
```

— *end example*] — *end note*]

1

[*Note:* Addresses of array elements and names or addresses of non-static class members are not acceptable *template-argument*s. [*Example:*

```
template<int* p> class X { };

int a[10];
```

```
struct S { int m; static int s; } s;

X<&a[2]> x3;                    // error: address of array element
X<&s.m> x4;                     // error: address of non-static member
X<&s.s> x5;                     // error: &S::s must be used
X<&S::s> x6;                    // OK: address of static member
```

— *end example*] — *end note*]

1

[*Note:* Temporaries, unnamed lvalues, and named lvalues with no linkage are not acceptable *template-arguments* when the corresponding *template-parameter* has reference type. [*Example:*

```
template<const int& CRI> struct B { /* ... */ };

B<1> b2;                        // error: temporary would be required for template argument

int c = 1;
B<c> b1;                        // OK
```

— *end example*] — *end note*]

1

The following conversions are performed on each expression used as a non-type *template-argument*. If a non-type *template-argument* cannot be converted to the type of the corresponding *template-parameter* then the program is ill-formed.

— For a non-type *template-parameter* of integral or enumeration type, conversions permitted in a converted constant expression (5.3643) are applied.

— for a non-type *template-parameter* of type pointer to object, qualification conversions (4.246) and the array-to-pointer conversion (4.180) are applied; if the *template-argument* is of type `std::nullptr_t`, the null pointer conversion (4.579) is applied. [*Note:* In particular, neither the null pointer conversion for a zero-valued integer literal (4.579) nor the derived-to-base conversion (4.579) are applied. Although `0` is a valid *template-argument* for a non-type *template-parameter* of integral type, it is not a valid *template-argument* for a non-type *template-parameter* of pointer type. However, both `(int*)0` and `nullptr` are valid *template-arguments* for a non-type *template-parameter* of type "pointer to int." — *end note*]

— For a non-type *template-parameter* of type reference to object, no conversions apply. The type referred to by the reference may be more cv-qualified than the (otherwise identical) type of the *template-argument*. The *template-parameter* is bound directly to the *template-argument*, which shall be an lvalue.

— For a non-type *template-parameter* of type pointer to function, the function-to-pointer conversion (4.220) is applied; if the *template-argument* is of type `std::nullptr_t`, the null pointer conversion (4.579) is applied. If the *template-argument* represents a set of overloaded functions (or a pointer to such), the matching function is selected from the set (13.1084).

— For a non-type *template-parameter* of type reference to function, no conversions apply. If the *template-argument* represents a set of overloaded functions, the matching function is selected from the set (13.1084).

— For a non-type *template-parameter* of type pointer to member function, if the *template-argument* is of type `std::nullptr_t`, the null member pointer conversion (4.710) is applied; otherwise, no conversions apply. If the *template-argument* represents a set of overloaded member functions, the matching member function is selected from the set (13.1084).

— For a non-type *template-parameter* of type pointer to data member, qualification conversions (4.246) are applied; if the *template-argument* is of type `std::nullptr_t`, the null member pointer conversion (4.710) is applied.

[*Example:*

```
template<const int* pci> struct X { /* ... */ };
int ai[10];
X<ai> xi;                       // array to pointer and qualification conversions

struct Y { /* ... */ };
template<const Y& b> struct Z { /* ... */ };
Y y;
Z<y> z;                         // no conversion, but note extra cv-qualification

template<int (&pa)[5]> struct W { /* ... */ };
int b[5];
W<b> w;                         // no conversion

void f(char);
void f(int);

template<void (*pf)(int)> struct A { /* ... */ };

A<&f> a;                        // selects f(int)
```

— *end example*]

14.574.1 Template template arguments [temp.arg.template]

1

A *template-argument* for a template *template-parameter* shall be the name of a class template or an alias template, expressed as *id-expression*. When the *template-argument* names a class template, only primary class templates are considered when matching the template template argument with the corresponding parameter; partial specializations are not considered even if their parameter lists match that of the template template parameter.

1 Any partial specializations (14.812.1) associated with the primary class template or primary variable template are considered when a specialization based on the template *template-parameter* is instantiated. If a specialization is not visible at the point of instantiation, and it would have been selected had it been visible, the program is ill-formed; no diagnostic is required. [*Example:*

```
template<class T> class A {      // primary template
  int x;
};
template<class T> class A<T*> { // partial specialization
  long x;
};
template<template<class U> class V> class C {
  V<int>  y;
  V<int*> z;
};
C<A> c;                          // V<int> within C<A> uses the primary template,
                                 // so c.y.x has type int
                                 // V<int*> within C<A> uses the partial specialization,
                                 // so c.z.x has type long
```

— *end example*]

2 A *template-argument* matches a template *template-parameter* (call it `P`) when each of the template parameters in the *template-parameter-list* of the *template-argument*'s corresponding class template or alias template (call it `A`) matches the corresponding template parameter in the *template-parameter-list* of `P`. Two template parameters match if they are of the same kind (type, non-type, template), for non-type *template-parameter*s, their types are equivalent (14.975.0.1), and for template *template-parameter*s, each of their corresponding *template-parameter*s matches, recursively. When `P`'s *template-parameter-list* contains a template parameter pack (14.703.1), the template parameter pack will match zero or more template parameters or template parameter packs in the *template-parameter-list* of `A` with the same type and form as the template parameter pack in `P` (ignoring whether those template parameters are template parameter packs).

[*Example:*

```
template<class T> class A { /* ... */ };
template<class T, class U = T> class B { /* ... */ };
template <class ... Types> class C { /* ... */ };

template<template<class> class P> class X { /* ... */ };
template<template<class ...> class Q> class Y { /* ... */ };

X<A> xa;            // OK
X<B> xb;            // ill-formed: default arguments for the parameters of a template argument are ignored
X<C> xc;            // ill-formed: a template parameter pack does not match a template parameter

Y<A> ya;            // OK
Y<B> yb;            // OK
Y<C> yc;            // OK
```

— *end example*]

[*Example:*

```
template <class T> struct eval;

template <template <class, class...> class TT, class T1, class... Rest>
struct eval<TT<T1, Rest...>> { };

template <class T1> struct A;
template <class T1, class T2> struct B;
template <int N> struct C;
template <class T1, int N> struct D;
template <class T1, class T2, int N = 17> struct E;

eval<A<int>> eA;                // OK: matches partial specialization of eval
eval<B<int, float>> eB;         // OK: matches partial specialization of eval
eval<C<17>> eC;                 // error: C does not match TT in partial specialization
eval<D<int, 17>> eD;            // error: D does not match TT in partial specialization
eval<E<int, float>> eE;         // error: E does not match TT in partial specialization
```

— *end example*]

14.595 Type equivalence [temp.type]

1

Two *template-id*s refer to the same class, function, or variable if

— their *template-name*s, *operator-function-id*s, or *literal-operator-id*s refer to the same template and

— their corresponding type *template-argument*s are the same type and

— their corresponding non-type template arguments of integral or enumeration type have identical values and

— their corresponding non-type *template-argument*s of pointer type refer to the same external object or function or are both the null pointer value and

— their corresponding non-type *template-argument*s of pointer-to-member type refer to the same class member or are both the null member pointer value and

— their corresponding non-type *template-argument*s of reference type refer to the same external object or function and

— their corresponding template *template-argument*s refer to the same template.

[*Example:*

```
template<class E, int size> class buffer { /* ... */ };
buffer<char,2*512> x;
buffer<char,1024> y;
```

declares `x` and `y` to be of the same type, and

```
template<class T, void(*err_fct)()> class list { /* ... */ };
list<int,&error_handler1> x1;
list<int,&error_handler2> x2;
list<int,&error_handler2> x3;
list<char,&error_handler2> x4;
```

declares `x2` and `x3` to be of the same type. Their type differs from the types of `x1` and `x4`.

```
template<class T> struct X { };
template<class> struct Y { };
template<class T> using Z = Y<T>;
X<Y<int> > y;
X<Z<int> > z;
```

declares `y` and `z` to be of the same type. — *end example*]

1

If an expression *e* involves a template parameter, `decltype(`*e*`)` denotes a unique dependent type. Two such *decltype-specifier*s refer to the same type only if their *expression*s are equivalent (14.975.0.1). [*Note:* however, it may be aliased, e.g., by a *typedef-name*. — *end note*]

14.616 Template declarations [temp.decls]

1 A *template-id*, that is, the *template-name* followed by a *template-argument-list* shall not be specified in the declaration of a primary template declaration. [*Example:*

```
template<class T1, class T2, int I> class A<T1, T2, I> { };      // error
template<class T1, int I> void sort<T1, I>(T1 data[I]);          // error
```

— *end example*] [*Note:* However, this syntax is allowed in class template partial specializations (14.812.1). — *end note*]

2 For purposes of name lookup and instantiation, default arguments and *exception-specification*s of function templates and default arguments and *exception-specification*s of member functions of class templates are considered definitions; each default argument or *exception-specification* is a separate definition which is unrelated to the function template definition or to any other default arguments or *exception-specification*s.

3

Because an *alias-declaration* cannot declare a *template-id*, it is not possible to partially or explicitly specialize an alias template.

14.619.1 Class templates [temp.class]

1 A class *template* defines the layout and operations for an unbounded set of related types. [*Example:* a single class template `List` might provide a common definition for list of `int`, list of `float`, and list of pointers to `Shape`s. — *end example*]

[*Example:* An array class template might be declared like this:

```
template<class T> class Array {
  T* v;
  int sz;
public:
  explicit Array(int);
  T& operator[](int);
  T& elem(int i) { return v[i]; }
};
```

2 The prefix `template <class T>` specifies that a template is being declared and that a *type-name* `T` will be used in the declaration. In other words, `Array` is a parameterized type with `T` as its parameter. — *end example*]

3

When a member function, a member class, a member enumeration, a static data member or a member template of a class template is defined outside of the class template definition, the member definition is defined as a template definition in which the *template-parameter*s are those of the class template. The names of the template parameters used in the definition of the member may be different from the template parameter names used in the class template definition. The template argument list following the class template name in the member definition shall name the parameters in the same order as the one used in the template parameter list of the member. Each template parameter pack shall be expanded with an ellipsis in the template argument list. [*Example:*

```
template<class T1, class T2> struct A {
  void f1();
  void f2();
};

template<class T2, class T1> void A<T2,T1>::f1() { }    // OK
template<class T2, class T1> void A<T1,T2>::f2() { }    // error

template<class ... Types> struct B {
  void f3();
  void f4();
};

template<class ... Types> void B<Types ...>::f3() { }    // OK
template<class ... Types> void B<Types>::f4() { }        // error
```

— *end example*]

1

In a redeclaration, partial specialization, explicit specialization or explicit instantiation of a class template, the *class-key* shall agree in kind with the original class template declaration (7.808.0.1).

14.630.0.1 Member functions of class templates [temp.mem.func]

1

A member function of a class template may be defined outside of the class template definition in which it is declared. [*Example:*

```
template<class T> class Array {
  T* v;
  int sz;
public:
  explicit Array(int);
  T& operator[](int);
  T& elem(int i) { return v[i]; }
};
```

declares three function templates. The subscript function might be defined like this:

```
template<class T> T& Array<T>::operator[](int i) {
  if (i<0 || sz<=i) error("Array: range error");
  return v[i];
}
```

— *end example*]

1

The *template-arguments* for a member function of a class template are determined by the *template-arguments* of the type of the object for which the member function is called. [*Example:* the *template-argument* for `Array<T>::operator[]()` will be determined by the `Array` to which the subscripting operation is applied.

```
Array<int> v1(20);
Array<dcomplex> v2(30);

v1[3] = 7;                      // Array<int>::operator[]()
v2[3] = dcomplex(7,8);          // Array<dcomplex>::operator[]()
```

— *end example*]

14.633.0.1 Member classes of class templates [temp.mem.class]

1

A member class of a class template may be defined outside the class template definition in which it is declared. [*Note:* The member class must be defined before its first use that requires an instantiation (14.1644.1). For example,

```
template<class T> struct A {
  class B;
};
A<int>::B* b1;                  // OK: requires A to be defined but not A::B
template<class T> class A<T>::B { };
A<int>::B  b2;                  // OK: requires A::B to be defined
```

— *end note*]

14.634.0.1 Static data members of class templates [temp.static]

1

A definition for a static data member or static data member template may be provided in a namespace scope enclosing the definition of the static member's class template. [*Example:*

```
template<class T> class X {
  static T s;
};
template<class T> T X<T>::s = 0;

struct limits {
  template<class T>
    static const T min;           // declaration
};

template<class T>
  const T limits::min = { };      // definition
```

— *end example*]

1 An explicit specialization of a static data member declared as an array of unknown bound can have a different bound from its definition, if any. [*Example:*

```
template <class T> struct A {
  static int i[];
};
template <class T> int A<T>::i[4];     // 4 elements
template <> int A<int>::i[] = { 1 };   // OK: 1 element
```

— *end example*]

14.640.0.1 Enumeration members of class templates [temp.mem.enum]

1

An enumeration member of a class template may be defined outside the class template definition. [*Example:*

```
template<class T> struct A {
  enum E : T;
};
A<int> a;
template<class T> enum A<T>::E : T { e1, e2 };
A<int>::E e = A<int>::e1;
```

— *end example*]

14.641.1 Member templates [temp.mem]

1

A template can be declared within a class or class template; such a template is called a member template. A member template can be defined within or outside its class definition or class template definition. A member template of a class template that is defined outside of its class template definition shall be specified with the *template-parameter*s of the class template followed by the *template-parameter*s of the member template. [*Example:*

```
template<class T> struct string {
  template<class T2> int compare(const T2&);
  template<class T2> string(const string<T2>& s) { /* ... */ }
```

```
};

template<class T> template<class T2> int string<T>::compare(const T2& s) {
}
```

— *end example*]

1

A local class of non-closure type shall not have member templates. Access control rules (Clause 11) apply to member template names. A destructor shall not be a member template. A non-template member function (8.360.1) with a given name and type and a member function template of the same name, which could be used to generate a specialization of the same type, can both be declared in a class. When both exist, a use of that name and type refers to the non-template member unless an explicit template argument list is supplied. [*Example:*

```
template <class T> struct A {
  void f(int);
  template <class T2> void f(T2);
};

template <> void A<int>::f(int) { }                    // non-template member function
template <> template <> void A<int>::f<>(int) { }      // member function template specialization

int main() {
  A<char> ac;
  ac.f(1);          // non-template
  ac.f('c');        // template
  ac.f<>(1);        // template
}
```

— *end example*]

1

A member function template shall not be virtual. [*Example:*

```
template <class T> struct AA {
  template <class C> virtual void g(C);   // error
  virtual void f();                       // OK
};
```

— *end example*]

1

A specialization of a member function template does not override a virtual function from a base class. [*Example:*

```
class B {
  virtual void f(int);
};

class D : public B {
  template <class T> void f(T); // does not override B::f(int)
  void f(int i) { f<>(i); }     // overriding function that calls
                                // the template instantiation
};
```

— *end example*]

1

A specialization of a conversion function template is referenced in the same way as a non-template conversion function that converts to the same type. [*Example:*

```
struct A {
  template <class T> operator T*();
};
template <class T> A::operator T*(){ return 0; }
template <> A::operator char*(){ return 0; }    // specialization
template A::operator void*();                   // explicit instantiation

int main() {
  A a;
  int* ip;
  ip = a.operator int*();        // explicit call to template operator
                                 // A::operator int*()
}
```

— *end example*] [*Note:* Because the explicit template argument list follows the function template name, and because conversion member function templates and constructor member function templates are called without using a function name, there is no way to provide an explicit template argument list for these function templates. — *end note*]

1

A specialization of a conversion function template is not found by name lookup. Instead, any conversion function templates visible in the context of the use are considered. For each such operator, if argument deduction succeeds (14.2413.0.1), the resulting specialization is used as if found by name lookup.

1 A *using-declaration* in a derived class cannot refer to a specialization of a conversion function template in a base class.

2

Overload resolution (13.946.0.1) and partial ordering (14.992.0.1) are used to select the best conversion function among multiple specializations of conversion function templates and/or non-template conversion functions.

14.703.1 Variadic templates [temp.variadic]

1

A *template parameter pack* is a template parameter that accepts zero or more template arguments. [*Example:*

```
template<class ... Types> struct Tuple { };

Tuple<> t0;                    // Types contains no arguments
Tuple<int> t1;                 // Types contains one argument: int
Tuple<int, float> t2;          // Types contains two arguments: int and float
Tuple<0> error;                // error: 0 is not a type
```

— *end example*]

1

A *function parameter pack* is a function parameter that accepts zero or more function arguments. [*Example:*

```
template<class ... Types> void f(Types ... args);
```

```
f();                // OK: args contains no arguments
f(1);               // OK: args contains one argument: int
f(2, 1.0);          // OK: args contains two arguments: int and double
```

— *end example*]

1

A *parameter pack* is either a template parameter pack or a function parameter pack.

1

A *pack expansion* consists of a *pattern* and an ellipsis, the instantiation of which produces zero or more instantiations of the pattern in a list (described below). The form of the pattern depends on the context in which the expansion occurs. Pack expansions can occur in the following contexts:

— In a function parameter pack (8.360.1); the pattern is the *parameter-declaration* without the ellipsis.

— In a template parameter pack that is a pack expansion (14.97):
  — if the template parameter pack is a *parameter-declaration*; the pattern is the *parameter-declaration* without the ellipsis;
  — if the template parameter pack is a *type-parameter* with a *template-parameter-list*; the pattern is the corresponding *type-parameter* without the ellipsis.

— In an *initializer-list* (8.821); the pattern is an *initializer-clause*.

— In a *base-specifier-list* (Clause 10); the pattern is a *base-specifier*.

— In a *mem-initializer-list* (12.1026.1) for a *mem-initializer* whose *mem-initializer-id* denotes a base class; the pattern is the *mem-initializer*.

— In a *template-argument-list* (14.326); the pattern is a *template-argument*.

— In a *dynamic-exception-specification* (15.228); the pattern is a *type-id*.

— In an *attribute-list* (7.1546.1); the pattern is an *attribute*.

— In an *alignment-specifier* (7.1556.1); the pattern is the *alignment-specifier* without the ellipsis.

— In a *capture-list* (5.509.1); the pattern is a *capture*.

— In a `sizeof...` expression (5.2075.1); the pattern is an *identifier*.

1

For the purpose of determining whether a parameter pack satisfies a rule regarding entities other than parameter packs, the parameter pack is considered to be the entity that would result from an instantiation of the pattern in which it appears.

[*Example:*

```
template<class ... Types> void f(Types ... rest);
template<class ... Types> void g(Types ... rest) {
  f(&rest ...);      // "&rest ..." is a pack expansion; "&rest" is its pattern
}
```

— *end example*]

1

A parameter pack whose name appears within the pattern of a pack expansion is expanded by that pack expansion. An appearance of the name of a parameter pack is only expanded by the innermost enclosing

pack expansion. The pattern of a pack expansion shall name one or more parameter packs that are not expanded by a nested pack expansion; such parameter packs are called *unexpanded* parameter packs in the pattern. All of the parameter packs expanded by a pack expansion shall have the same number of arguments specified. An appearance of a name of a parameter pack that is not expanded is ill-formed. [*Example:*

```
template<typename...> struct Tuple {};
template<typename T1, typename T2> struct Pair {};

template<class ... Args1> struct zip {
  template<class ... Args2> struct with {
    typedef Tuple<Pair<Args1, Args2> ... > type;
  };
};

typedef zip<short, int>::with<unsigned short, unsigned>::type T1;
    // T1 is Tuple<Pair<short, unsigned short>, Pair<int, unsigned>>
typedef zip<short>::with<unsigned short, unsigned>::type T2;
    // error: different number of arguments specified for Args1 and Args2

template<class ... Args>
  void g(Args ... args) {                // OK: Args is expanded by the function parameter pack args
    f(const_cast<const Args*>(&args)...); // OK: "Args" and "args" are expanded
    f(5 ...);                             // error: pattern does not contain any parameter packs
    f(args);                              // error: parameter pack "args" is not expanded
    f(h(args ...) + args ...);            // OK: first "args" expanded within h, second
                                          // "args" expanded within f
  }
```

— *end example*]

1

The instantiation of a pack expansion that is not a `sizeof...` expression produces a list $\mathtt{E}_1, \mathtt{E}_2, ..., \mathtt{E}_N$, where N is the number of elements in the pack expansion parameters. Each $\mathtt{E}_i$ is generated by instantiating the pattern and replacing each pack expansion parameter with its ith element. Such an element, in the context of the instantiation, is interpreted as follows:

— if the pack is a template parameter pack, the element is a template parameter (14.97) of the corresponding kind (type or non-type) designating the type or value from the template argument; otherwise,

— if the pack is a function parameter pack, the element is an *id-expression* designating the function parameter that resulted from the instantiation of the pattern where the pack is declared.

All of the $\mathtt{E}_i$ become elements in the enclosing list. [*Note:* The variety of list varies with the context: *expression-list*, *base-specifier-list*, *template-argument-list*, etc. — *end note*] When N is zero, the instantiation of the expansion produces an empty list. Such an instantiation does not alter the syntactic interpretation of the enclosing construct, even in cases where omitting the list entirely would otherwise be ill-formed or would result in an ambiguity in the grammar. [*Example:*

```
template<class... T> struct X : T... { };
template<class... T> void f(T... values) {
  X<T...> x(values...);
}

template void f<>();  // OK: X<> has no base classes
                      // x is a variable of type X<> that is value-initialized
```

— *end example*]

1 The instantiation of a `sizeof...` expression (5.2075.1) produces an integral constant containing the number of elements in the parameter pack it expands.

14.755.1 Friends [temp.friend]

1

A friend of a class or class template can be a function template or class template, a specialization of a function template or class template, or a non-template function or class. For a friend function declaration that is not a template declaration:

— if the name of the friend is a qualified or unqualified *template-id*, the friend declaration refers to a specialization of a function template, otherwise,

— if the name of the friend is a *qualified-id* and a matching non-template function is found in the specified class or namespace, the friend declaration refers to that function, otherwise,

— if the name of the friend is a *qualified-id* and a matching function template is found in the specified class or namespace, the friend declaration refers to the deduced specialization of that function template (14.2655.0.1), otherwise,

— the name shall be an *unqualified-id* that declares (or redeclares) a non-template function.

[*Example:*

```
template<class T> class task;
template<class T> task<T>* preempt(task<T>*);

template<class T> class task {
  friend void next_time();
  friend void process(task<T>*);
  friend task<T>* preempt<T>(task<T>*);
  template<class C> friend int func(C);

  friend class task<int>;
  template<class P> friend class frd;
};
```

Here, each specialization of the `task` class template has the function `next_time` as a friend; because `process` does not have explicit *template-arguments*, each specialization of the `task` class template has an appropriately typed function `process` as a friend, and this friend is not a function template specialization; because the friend `preempt` has an explicit *template-argument* `T`, each specialization of the `task` class template has the appropriate specialization of the function template `preempt` as a friend; and each specialization of the `task` class template has all specializations of the function template `func` as friends. Similarly, each specialization of the `task` class template has the class template specialization `task<int>` as a friend, and has all specializations of the class template `frd` as friends. — *end example*]

1

A friend template may be declared within a class or class template. A friend function template may be defined within a class or class template, but a friend class template may not be defined in a class or class template. In these cases, all specializations of the friend class or friend function template are friends of the class or class template granting friendship. [*Example:*

```
class A {
  template<class T> friend class B;                // OK
  template<class T> friend void f(T){ /* ...  */ }  // OK
};
```

— *end example*]

1 A template friend declaration specifies that all specializations of that template, whether they are implicitly instantiated (14.1644.1), partially specialized (14.812.1) or explicitly specialized (14.1921.1), are friends of the class containing the template friend declaration. [*Example:*

```
class X {
  template<class T> friend struct A;
  class Y { };
};

template<class T> struct A { X::Y ab; };            // OK
template<class T> struct A<T*> { X::Y ab; };        // OK
```

— *end example*]

2

When a function is defined in a friend function declaration in a class template, the function is instantiated when the function is odr-used (3.91). The same restrictions on multiple declarations and definitions that apply to non-template function declarations and definitions also apply to these implicit definitions.

1

A member of a class template may be declared to be a friend of a non-template class. In this case, the corresponding member of every specialization of the class template is a friend of the class granting friendship. For explicit specializations the corresponding member is the member (if any) that has the same name, kind (type, function, class template, or function template), template parameters, and signature as the member of the class template instantiation that would otherwise have been generated. [*Example:*

```
template<class T> struct A {
  struct B { };
  void f();
  struct D {
    void g();
  };
};
template<> struct A<int> {
  struct B { };
  int f();
  struct D {
    void g();
  };
};

class C {
  template<class T> friend struct A<T>::B;     // grants friendship to A<int>::B even though
                                               // it is not a specialization of A<T>::B
  template<class T> friend void A<T>::f();     // does not grant friendship to A<int>::f()
                                               // because its return type does not match
  template<class T> friend void A<T>::D::g(); // does not grant friendship to A<int>::D::g()
                                               // because A<int>::D is not a specialization of A<T>::D
};
```

— *end example*]

1 [*Note:* A friend declaration may first declare a member of an enclosing namespace scope (14.1608.1). — *end note*]

2

A friend template shall not be declared in a local class.

1

Friend declarations shall not declare partial specializations. [*Example:*

```
template<class T> class A { };
class X {
  template<class T> friend class A<T*>; // error
};
```

— *end example*]

1

When a friend declaration refers to a specialization of a function template, the function parameter declarations shall not include default arguments, nor shall the inline specifier be used in such a declaration.

14.812.1 Class template partial specializations [temp.class.spec]

1

A *primary* class template declaration is one in which the class template name is an identifier. A template declaration in which the class template name is a *simple-template-id* is a *partial specialization* of the class template named in the *simple-template-id*. A partial specialization of a class template provides an alternative definition of the template that is used instead of the primary definition when the arguments in a specialization match those given in the partial specialization (14.872.0.1). The primary template shall be declared before any specializations of that template. A partial specialization shall be declared before the first use of a class template specialization that would make use of the partial specialization as the result of an implicit or explicit instantiation in every translation unit in which such a use occurs; no diagnostic is required.

1

Each class template partial specialization is a distinct template and definitions shall be provided for the members of a template partial specialization (14.947.0.1).

1

[*Example:*

```
template<class T1, class T2, int I> class A             { };    // #1
template<class T, int I>            class A<T, T*, I>   { };    // #2
template<class T1, class T2, int I> class A<T1*, T2, I> { };    // #3
template<class T>                   class A<int, T*, 5> { };    // #4
template<class T1, class T2, int I> class A<T1, T2*, I> { };    // #5
```

The first declaration declares the primary (unspecialized) class template. The second and subsequent declarations declare partial specializations of the primary template. — *end example*]

1

The template parameters are specified in the angle bracket enclosed list that immediately follows the keyword `template`. For partial specializations, the template argument list is explicitly written immediately following the class template name. For primary templates, this list is implicitly described by the template parameter list. Specifically, the order of the template arguments is the sequence in which they appear in the template parameter list. [*Example:* the template argument list for the primary template in the example above is `<T1, T2, I>`. — *end example*] [*Note:* The template argument list shall not be specified in the primary template declaration. For example,

```
template<class T1, class T2, int I> class A<T1, T2, I>  { };    // error
```

— *end note*]

1

A class template partial specialization may be declared or redeclared in any namespace scope in which its definition may be defined (14.619.1 and 14.641.1). [*Example:*

```
template<class T> struct A {
  struct C {
    template<class T2> struct B { };
  };
};

// partial specialization of A<T>::C::B<T2>
template<class T> template<class T2>
  struct A<T>::C::B<T2*> { };

A<short>::C::B<int*> absip;      // uses partial specialization
```

— *end example*]

1

Partial specialization declarations themselves are not found by name lookup. Rather, when the primary template name is used, any previously-declared partial specializations of the primary template are also considered. One consequence is that a *using-declaration* which refers to a class template does not restrict the set of partial specializations which may be found through the *using-declaration*. [*Example:*

```
namespace N {
  template<class T1, class T2> class A { };          // primary template
}

using N::A;                                  // refers to the primary template

namespace N {
  template<class T> class A<T, T*> { }; // partial specialization
}

A<int,int*> a;                     // uses the partial specialization, which is found through
                                   // the using declaration which refers to the primary template
```

— *end example*]

1

A non-type argument is non-specialized if it is the name of a non-type parameter. All other non-type arguments are specialized.

1

Within the argument list of a class template partial specialization, the following restrictions apply:

— A partially specialized non-type argument expression shall not involve a template parameter of the partial specialization except when the argument expression is a simple *identifier*. [*Example:*

```
template <int I, int J> struct A {};
template <int I> struct A<I+5, I*2> {}; // error

template <int I, int J> struct B {};
template <int I> struct B<I, I> {};     // OK
```

— *end example*]

— The type of a template parameter corresponding to a specialized non-type argument shall not be dependent on a parameter of the specialization. [*Example:*

```
template <class T, T t> struct C {};
template <class T> struct C<T, 1>;                  // error

template< int X, int (*array_ptr)[X] > class A {};
int array[5];
template< int X > class A<X,&array> { };            // error
```

— *end example*]

— The argument list of the specialization shall not be identical to the implicit argument list of the primary template.

— The specialization shall be more specialized than the primary template (14.919.0.1).

— The template parameter list of a specialization shall not contain default template argument values.[140]

— An argument shall not contain an unexpanded parameter pack. If an argument is a pack expansion (14.703.1), it shall be the last argument in the template argument list.

14.872.0.1 Matching of class template partial specializations [temp.class.spec.match]

1

When a class template is used in a context that requires an instantiation of the class, it is necessary to determine whether the instantiation is to be generated using the primary template or one of the partial specializations. This is done by matching the template arguments of the class template specialization with the template argument lists of the partial specializations.

— If exactly one matching specialization is found, the instantiation is generated from that specialization.

— If more than one matching specialization is found, the partial order rules (14.919.0.1) are used to determine whether one of the specializations is more specialized than the others. If none of the specializations is more specialized than all of the other matching specializations, then the use of the class template is ambiguous and the program is ill-formed.

— If no matches are found, the instantiation is generated from the primary template.

1

A partial specialization matches a given actual template argument list if the template arguments of the partial specialization can be deduced from the actual template argument list (14.2114.1). [*Example:*

```
A<int, int, 1>   a1;          // uses #1
A<int, int*, 1>  a2;          // uses #2, T is int, I is 1
A<int, char*, 5> a3;          // uses #4, T is char
A<int, char*, 1> a4;          // uses #5, T1 is int, T2 is char, I is 1
A<int*, int*, 2> a5;          // ambiguous: matches #3 and #5
```

— *end example*]

1

A non-type template argument can also be deduced from the value of an actual template argument of a non-type parameter of the primary template. [*Example:* the declaration of `a2` above. — *end example*]

140) There is no way in which they could be used.

1

In a type name that refers to a class template specialization, (e.g., `A<int, int, 1>`) the argument list shall match the template parameter list of the primary template. The template arguments of a specialization are deduced from the arguments of the primary template.

14.919.0.1 Partial ordering of class template specializations [temp.class.order]

1

For two class template partial specializations, the first is at least as specialized as the second if, given the following rewrite to two function templates, the first function template is at least as specialized as the second according to the ordering rules for function templates (14.992.0.1):

— the first function template has the same template parameters as the first partial specialization and has a single function parameter whose type is a class template specialization with the template arguments of the first partial specialization, and

— the second function template has the same template parameters as the second partial specialization and has a single function parameter whose type is a class template specialization with the template arguments of the second partial specialization.

1 [*Example:*

```
template<int I, int J, class T> class X { };
template<int I, int J>          class X<I, J, int> { }; // #1
template<int I>                 class X<I, I, int> { }; // #2

template<int I, int J> void f(X<I, J, int>);            // A
template<int I>        void f(X<I, I, int>);            // B
```

The partial specialization `#2` is more specialized than the partial specialization `#1` because the function template `B` is more specialized than the function template `A` according to the ordering rules for function templates. — *end example*]

14.947.0.1 Members of class template specializations [temp.class.spec.mfunc]

1

The template parameter list of a member of a class template partial specialization shall match the template parameter list of the class template partial specialization. The template argument list of a member of a class template partial specialization shall match the template argument list of the class template partial specialization. A class template specialization is a distinct template. The members of the class template partial specialization are unrelated to the members of the primary template. Class template partial specialization members that are used in a way that requires a definition shall be defined; the definitions of members of the primary template are never used as definitions for members of a class template partial specialization. An explicit specialization of a member of a class template partial specialization is declared in the same way as an explicit specialization of the primary template. [*Example:*

```
// primary template
template<class T, int I> struct A {
  void f();
};

template<class T, int I> void A<T,I>::f() { }

// class template partial specialization
template<class T> struct A<T,2> {
  void f();
```

```
  void g();
  void h();
};

// member of class template partial specialization
template<class T> void A<T,2>::g() { }

// explicit specialization
template<> void A<char,2>::h() { }

int main() {
  A<char,0> a0;
  A<char,2> a2;
  a0.f();                       // OK, uses definition of primary template's member
  a2.g();                       // OK, uses definition of
                                // partial specialization's member
  a2.h();                       // OK, uses definition of
                                // explicit specialization's member
  a2.f();                       // ill-formed, no definition of f for A<T,2>
                                // the primary template is not used here
}
```

— *end example*]

1

If a member template of a class template is partially specialized, the member template partial specializations are member templates of the enclosing class template; if the enclosing class template is instantiated (14.1644.1, 14.1841.1), a declaration for every member template partial specialization is also instantiated as part of creating the members of the class template specialization. If the primary member template is explicitly specialized for a given (implicit) specialization of the enclosing class template, the partial specializations of the member template are ignored for this specialization of the enclosing class template. If a partial specialization of the member template is explicitly specialized for a given (implicit) specialization of the enclosing class template, the primary member template and its other partial specializations are still considered for this specialization of the enclosing class template. [*Example:*

```
template<class T> struct A {
  template<class T2> struct B {};                    // #1
  template<class T2> struct B<T2*> {};               // #2
};

template<> template<class T2> struct A<short>::B {};  // #3

A<char>::B<int*>  abcip;  // uses #2
A<short>::B<int*> absip;  // uses #3
A<char>::B<int>  abci;    // uses #1
```

— *end example*]

14.972.1 Function templates [temp.fct]

1 A function template defines an unbounded set of related functions. [*Example:* a family of sort functions might be declared like this:

```
template<class T> class Array { };
template<class T> void sort(Array<T>&);
```

— *end example*]

2

A function template can be overloaded with other function templates and with non-template functions (8.360.1). A non-template function is not related to a function template (i.e., it is never considered to be a specialization), even if it has the same name and type as a potentially generated function template specialization.[141]

14.975.0.1 Function template overloading [temp.over.link]

1

It is possible to overload function templates so that two different function template specializations have the same type. [*Example:*

```
// file1.c
template<class T>
  void f(T*);
void g(int* p) {
  f(p); // calls f<int>(int*)
}
```

```
// file2.c
template<class T>
  void f(T);
void h(int* p) {
  f(p); // calls f<int*>(int*)
}
```

— *end example*]

1

Such specializations are distinct functions and do not violate the one definition rule (3.91).

1

The signature of a function template is defined in 1.27. The names of the template parameters are significant only for establishing the relationship between the template parameters and the rest of the signature. [*Note:* Two distinct function templates may have identical function return types and function parameter lists, even if overload resolution alone cannot distinguish them.

```
template<class T> void f();
template<int I> void f();        // OK: overloads the first template
                                 // distinguishable with an explicit template argument list
```

— *end note*]

1

When an expression that references a template parameter is used in the function parameter list or the return type in the declaration of a function template, the expression that references the template parameter is part of the signature of the function template. This is necessary to permit a declaration of a function template in one translation unit to be linked with another declaration of the function template in another translation unit and, conversely, to ensure that function templates that are intended to be distinct are not linked with one another. [*Example:*

```
template <int I, int J> A<I+J> f(A<I>, A<J>);    // #1
template <int K, int L> A<K+L> f(A<K>, A<L>);    // same as #1
template <int I, int J> A<I-J> f(A<I>, A<J>);    // different from #1
```

— *end example*] [*Note:* Most expressions that use template parameters use non-type template parameters, but it is possible for an expression to reference a type parameter. For example, a template type parameter can be used in the `sizeof` operator. — *end note*]

141) That is, declarations of non-template functions do not merely guide overload resolution of function template specializations with the same name. If such a non-template function is odr-used (3.91) in a program, it must be defined; it will not be implicitly instantiated using the function template definition.

1

Two expressions involving template parameters are considered *equivalent* if two function definitions containing the expressions would satisfy the one definition rule (3.91), except that the tokens used to name the template parameters may differ as long as a token used to name a template parameter in one expression is replaced by another token that names the same template parameter in the other expression. For determining whether two dependent names (14.1257.1) are equivalent, only the name itself is considered, not the result of name lookup in the context of the template. If multiple declarations of the same function template differ in the result of this name lookup, the result for the first declaration is used. [*Example:*

```
template <int I, int J> void f(A<I+J>);          // #1
template <int K, int L> void f(A<K+L>);          // same as #1

template <class T> decltype(g(T())) h();
int g(int);
template <class T> decltype(g(T())) h()          // redeclaration of h() uses the earlier lookup
  { return g(T()); }                             // ...although the lookup here does find g(int)
int i = h<int>();                                // template argument substitution fails; g(int)
                                                 // was not in scope at the first declaration of h()
```

— *end example*] Two expressions involving template parameters that are not equivalent are *functionally equivalent* if, for any given set of template arguments, the evaluation of the expression results in the same value.

1

Two function templates are *equivalent* if they are declared in the same scope, have the same name, have identical template parameter lists, and have return types and parameter lists that are equivalent using the rules described above to compare expressions involving template parameters. Two function templates are *functionally equivalent* if they are equivalent except that one or more expressions that involve template parameters in the return types and parameter lists are functionally equivalent using the rules described above to compare expressions involving template parameters. If a program contains declarations of function templates that are functionally equivalent but not equivalent, the program is ill-formed; no diagnostic is required.

1 [*Note:* This rule guarantees that equivalent declarations will be linked with one another, while not requiring implementations to use heroic efforts to guarantee that functionally equivalent declarations will be treated as distinct. For example, the last two declarations are functionally equivalent and would cause a program to be ill-formed:

```
// Guaranteed to be the same
template <int I> void f(A<I>, A<I+10>);
template <int I> void f(A<I>, A<I+10>);

// Guaranteed to be different
template <int I> void f(A<I>, A<I+10>);
template <int I> void f(A<I>, A<I+11>);

// Ill-formed, no diagnostic required
template <int I> void f(A<I>, A<I+10>);
template <int I> void f(A<I>, A<I+1+2+3+4>);
```

— *end note*]

14.992.0.1 Partial ordering of function templates [temp.func.order]

1

If a function template is overloaded, the use of a function template specialization might be ambiguous because template argument deduction (14.2114.1) may associate the function template specialization with more than one function template declaration. *Partial ordering* of overloaded function template declarations is used in the following contexts to select the function template to which a function template specialization refers:

— during overload resolution for a call to a function template specialization (13.562.1);

— when the address of a function template specialization is taken;

— when a placement operator delete that is a function template specialization is selected to match a placement operator new (3.2428.0.1, 5.2221.1);

— when a friend function declaration (14.755.1), an explicit instantiation (14.1841.1) or an explicit specialization (14.1921.1) refers to a function template specialization.

1

Partial ordering selects which of two function templates is more specialized than the other by transforming each template in turn (see next paragraph) and performing template argument deduction using the function type. The deduction process determines whether one of the templates is more specialized than the other. If so, the more specialized template is the one chosen by the partial ordering process.

1

To produce the transformed template, for each type, non-type, or template template parameter (including template parameter packs (14.703.1) thereof) synthesize a unique type, value, or class template respectively and substitute it for each occurrence of that parameter in the function type of the template. If only one of the function templates is a non-static member of some class `A`, that function template is considered to have a new first parameter inserted in its function parameter list. Given *cv* as the cv-qualifiers of the function template (if any), the new parameter is of type "rvalue reference to *cv* `A`" if the optional *ref-qualifier* of the function template is `&&`, or of type "lvalue reference to *cv* `A`" otherwise. [*Note:* This allows a non-static member to be ordered with respect to a nonmember function and for the results to be equivalent to the ordering of two equivalent nonmembers. — *end note*] [*Example:*

```
struct A { };
template<class T> struct B {
  template<class R> int operator*(R&);              // #1
};

template<class T, class R> int operator*(T&, R&);   // #2

// The declaration of B::operator* is transformed into the equivalent of
// template<class R> int operator*(B<A>&, R&);      // #1a

int main() {
  A a;
  B<A> b;
  b * a;                                            // calls #1a
}
```

— *end example*]

1

Using the transformed function template's function type, perform type deduction against the other template as described in 14.2418.0.1.

[*Example:*

```
template<class T> struct A { A(); };

template<class T> void f(T);
template<class T> void f(T*);
template<class T> void f(const T*);

template<class T> void g(T);
template<class T> void g(T&);

template<class T> void h(const T&);
template<class T> void h(A<T>&);

void m() {
  const int* p;
  f(p);             // f(const T*) is more specialized than f(T) or f(T*)
  float x;
  g(x);             // Ambiguous: g(T) or g(T&)
  A<int> z;
  h(z);             // overload resolution selects h(A<T>&)
  const A<int> z2;
  h(z2);            // h(const T&) is called because h(A<T>&) is not callable
}
```

— *end example*]

1

[*Note:* Since partial ordering in a call context considers only parameters for which there are explicit call arguments, some parameters are ignored (namely, function parameter packs, parameters with default arguments, and ellipsis parameters). [*Example:*

```
template<class T> void f(T);            // #1
template<class T> void f(T*, int=1);    // #2
template<class T> void g(T);            // #3
template<class T> void g(T*, ...);      // #4

int main() {
  int* ip;
  f(ip);            // calls #2
  g(ip);            // calls #4
}
```

— *end example*] [*Example:*

```
template<class T, class U> struct A { };

template<class T, class U> void f(U, A<U, T>* p = 0); // #1
template<         class U> void f(U, A<U, U>* p = 0); // #2
template<class T         > void g(T, T = T());        // #3
template<class T, class... U> void g(T, U ...);       // #4

void h() {
  f<int>(42, (A<int, int>*)0);                         // calls #2
  f<int>(42);                                          // error: ambiguous
  g(42);                                               // error: ambiguous
}
```

— *end example*] [*Example:*

```
template<class T, class... U> void f(T, U...);        // #1
template<class T            > void f(T);              // #2
template<class T, class... U> void g(T*, U...);       // #3
template<class T            > void g(T);              // #4

void h(int i) {
  f(&i);                                              // error: ambiguous
  g(&i);                                              // OK: calls #3
}
```

— *end example*] — *end note*]

14.1031.1 Alias templates [temp.alias]

1

A *template-declaration* in which the *declaration* is an *alias-declaration* (Clause 7) declares the *identifier* to be a *alias template*. An alias template is a name for a family of types. The name of the alias template is a *template-name*.

1

When a *template-id* refers to the specialization of an alias template, it is equivalent to the associated type obtained by substitution of its *template-arguments* for the *template-parameters* in the *type-id* of the alias template. [*Note:* An alias template name is never deduced. — *end note*] [*Example:*

```
template<class T> struct Alloc { /* ... */ };
template<class T> using Vec = vector<T, Alloc<T>>;
Vec<int> v;         // same as vector<int, Alloc<int>> v;

template<class T>
  void process(Vec<T>& v)
  { /* ... */ }

template<class T>
  void process(vector<T, Alloc<T>>& w)
  { /* ... */ }     // error: redefinition

template<template<class> class TT>
  void f(TT<int>);

f(v);               // error: Vec not deduced

template<template<class,class> class TT>
  void g(TT<int, Alloc<int>>);
g(v);               // OK: TT = vector
```

— *end example*]

1

The *type-id* in an alias template declaration shall not refer to the alias template being declared. The type produced by an alias template specialization shall not directly or indirectly make use of that specialization. [*Example:*

```
template <class T> struct A;
template <class T> using B = typename A<T>::U;
template <class T> struct A {
  typedef B<T> U;
```

```
};
B<short> b;          // error: instantiation of B<short> uses own type via A<short>::U
```

— *end example*]

14.1061 Name resolution [temp.res]

1

Three kinds of names can be used within a template definition:

— The name of the template itself, and names declared within the template itself.

— Names dependent on a *template-parameter* (14.1257.1).

— Names from scopes which are visible within the template definition.

1

A name used in a template declaration or definition and that is dependent on a *template-parameter* is assumed not to name a type unless the applicable name lookup finds a type name or the name is qualified by the keyword `typename`. [*Example:*

```
// no B declared here

class X;

template<class T> class Y {
  class Z;                      // forward declaration of member class

  void f() {
    X* a1;                      // declare pointer to X
    T* a2;                      // declare pointer to T
    Y* a3;                      // declare pointer to Y<T>
    Z* a4;                      // declare pointer to Z
    typedef typename T::A TA;
    TA* a5;                     // declare pointer to T's A
    typename T::A* a6;          // declare pointer to T's A
    T::A* a7;                   // T::A is not a type name:
                                // multiply T::A by a7; ill-formed,
                                // no visible declaration of a7
    B* a8;                      // B is not a type name:
                                // multiply B by a8; ill-formed,
                                // no visible declarations of B and a8
  }
};
```

— *end example*]

1

When a *qualified-id* is intended to refer to a type that is not a member of the current instantiation (14.1319.0.1) and its *nested-name-specifier* refers to a dependent type, it shall be prefixed by the keyword `typename`, forming a *typename-specifier*. If the *qualified-id* in a *typename-specifier* does not denote a type, the program is ill-formed.

typename-specifier:
  `typename` *nested-name-specifier identifier*
  `typename` *nested-name-specifier* `template`$_{opt}$ *simple-template-id*

1

If a specialization of a template is instantiated for a set of *template-arguments* such that the *qualified-id* prefixed by `typename` does not denote a type, the specialization is ill-formed. The usual qualified name lookup (3.1236.1) is used to find the *qualified-id* even in the presence of `typename`. [*Example:*

```
struct A {
  struct X { };
  int X;
};
struct B {
  struct X { };
};
template<class T> void f(T t) {
  typename T::X x;
}
void foo() {
  A a;
  B b;
  f(b);              // OK: T::X refers to B::X
  f(a);              // error: T::X refers to the data member A::X not the struct A::X
}
```

— *end example*]

1

A qualified name used as the name in a *mem-initializer-id*, a *base-specifier*, or an *elaborated-type-specifier* is implicitly assumed to name a type, without the use of the `typename` keyword. In a *nested-name-specifier* that immediately contains a *nested-name-specifier* that depends on a template parameter, the *identifier* or *simple-template-id* is implicitly assumed to name a type, without the use of the `typename` keyword. [*Note:* The `typename` keyword is not permitted by the syntax of these constructs. — *end note*]

1 If, for a given set of template arguments, a specialization of a template is instantiated that refers to a *qualified-id* that denotes a type, and the *qualified-id* refers to a member of an unknown specialization, the *qualified-id* shall either be prefixed by `typename` or shall be used in a context in which it implicitly names a type as described above. [*Example:*

```
template <class T> void f(int i) {
  T::x * i;          // T::x must not be a type
}

struct Foo {
  typedef int x;
};

struct Bar {
  static int const x = 5;
};

int main() {
  f<Bar>(1);         // OK
  f<Foo>(1);         // error: Foo::x is a type
}
```

— *end example*]

2

Within the definition of a class template or within the definition of a member of a class template following

the *declarator-id*, the keyword `typename` is not required when referring to the name of a previously declared member of the class template that declares a type. [*Note:* such names can be found using unqualified name lookup (3.952.1), class member lookup (3.1282.0.1) into the current instantiation (14.1319.0.1), or class member access expression lookup (3.1423.1) when the type of the object expression is the current instantiation (14.1390.0.1). — *end note*] [*Example:*

```
template<class T> struct A {
  typedef int B;
  B b;              // OK, no typename required
};
```

— *end example*]

1

Knowing which names are type names allows the syntax of every template to be checked. No diagnostic shall be issued for a template for which a valid specialization can be generated. If no valid specialization can be generated for a template, and that template is not instantiated, the template is ill-formed, no diagnostic required. If every valid specialization of a variadic template requires an empty template parameter pack, the template is ill-formed, no diagnostic required. If a type used in a non-dependent name is incomplete at the point at which a template is defined but is complete at the point at which an instantiation is done, and if the completeness of that type affects whether or not the program is well-formed or affects the semantics of the program, the program is ill-formed; no diagnostic is required. [*Note:* If a template is instantiated, errors will be diagnosed according to the other rules in this Standard. Exactly when these errors are diagnosed is a quality of implementation issue. — *end note*] [*Example:*

```
int j;
template<class T> class X {
  void f(T t, int i, char* p) {
    t = i;          // diagnosed if X::f is instantiated
                    // and the assignment to t is an error
    p = i;          // may be diagnosed even if X::f is
                    // not instantiated
    p = j;          // may be diagnosed even if X::f is
                    // not instantiated
  }
  void g(T t) {
    +;              // may be diagnosed even if X::g is
                    // not instantiated
  }
};

template<class... T> struct A {
  void operator++(int, T... t);                 // error: too many parameters
};
template<class... T> union X : T... { };        // error: union with base class
template<class... T> struct A : T...,  T... { };// error: duplicate base class
```

— *end example*]

1

When looking for the declaration of a name used in a template definition, the usual lookup rules (3.952.1, 3.1092.1) are used for non-dependent names. The lookup of names dependent on the template parameters is postponed until the actual template argument is known (14.1257.1). [*Example:*

```
#include <iostream>
using namespace std;
```

```
template<class T> class Set {
  T* p;
  int cnt;
public:
  Set();
  Set<T>(const Set<T>&);
  void printall() {
    for (int i = 0; i<cnt; i++)
      cout << p[i] << '\n';
  }
};
```

in the example, `i` is the local variable `i` declared in `printall`, `cnt` is the member `cnt` declared in `Set`, and `cout` is the standard output stream declared in `iostream`. However, not every declaration can be found this way; the resolution of some names must be postponed until the actual *template-arguments* are known. For example, even though the name `operator<<` is known within the definition of `printall()` and a declaration of it can be found in `<iostream>`, the actual declaration of `operator<<` needed to print `p[i]` cannot be known until it is known what type `T` is (14.1257.1). — *end example*]

1

If a name does not depend on a *template-parameter* (as defined in 14.1257.1), a declaration (or set of declarations) for that name shall be in scope at the point where the name appears in the template definition; the name is bound to the declaration (or declarations) found at that point and this binding is not affected by declarations that are visible at the point of instantiation. [*Example:*

```
void f(char);

template<class T> void g(T t) {
  f(1);             // f(char)
  f(T(1));          // dependent
  f(t);             // dependent
  dd++;             // not dependent
                    // error: declaration for dd not found
}

enum E { e };
void f(E);

double dd;
void h() {
  g(e);             // will cause one call of f(char) followed
                    // by two calls of f(E)
  g('a');           // will cause three calls of f(char)
}
```

— *end example*]

1 [*Note:* For purposes of name lookup, default arguments and *exception-specifications* of function templates and default arguments and *exception-specifications* of member functions of class templates are considered definitions (14.616). — *end note*]

14.1177.1 Locally declared names [temp.local]

1

Like normal (non-template) classes, class templates have an injected-class-name (Clause 9). The injected-class-name can be used as a *template-name* or a *type-name*. When it is used with a *template-argument-list*, as a *template-argument* for a template *template-parameter*, or as the final identifier in the *elaborated-type-specifier* of a friend class template declaration, it refers to the class template itself. Otherwise, it is equivalent to the *template-name* followed by the *template-parameter*s of the class template enclosed in `<>`.

1

Within the scope of a class template specialization or partial specialization, when the injected-class-name is used as a *type-name*, it is equivalent to the *template-name* followed by the *template-argument*s of the class template specialization or partial specialization enclosed in `<>`. [*Example:*

```
template<template<class> class T> class A { };
template<class T> class Y;
template<> class Y<int> {
  Y* p;                              // meaning Y<int>
  Y<char>* q;                        // meaning Y<char>
  A<Y>* a;                           // meaning A<::Y>
  class B {
    template<class> friend class Y;  // meaning ::Y
  };
};
```

— *end example*]

1

The injected-class-name of a class template or class template specialization can be used either as a *template-name* or a *type-name* wherever it is in scope. [*Example:*

```
template <class T> struct Base {
  Base* p;
};

template <class T> struct Derived: public Base<T> {
  typename Derived::Base* p;    // meaning Derived::Base<T>
};

template<class T, template<class> class U = T::template Base> struct Third { };
Third<Base<int> > t;            // OK: default argument uses injected-class-name as a template
```

— *end example*]

1

A lookup that finds an injected-class-name (10.52) can result in an ambiguity in certain cases (for example, if it is found in more than one base class). If all of the injected-class-names that are found refer to specializations of the same class template, and if the name is used as a *template-name*, the reference refers to the class template itself and not a specialization thereof, and is not ambiguous. [*Example:*

```
template <class T> struct Base { };
template <class T> struct Derived: Base<int>, Base<char> {
  typename Derived::Base b;             // error: ambiguous
  typename Derived::Base<double> d;     // OK
};
```

— *end example*]

1

When the normal name of the template (i.e., the name from the enclosing scope, not the injected-class-name) is used, it always refers to the class template itself and not a specialization of the template. [*Example:*

```
template<class T> class X {
  X* p;             // meaning X<T>
  X<T>* p2;
  X<int>* p3;
  ::X* p4;          // error: missing template argument list
                    // ::X does not refer to the injected-class-name
};
```

— *end example*]

1

A *template-parameter* shall not be redeclared within its scope (including nested scopes). A *template-parameter* shall not have the same name as the template name. [*Example:*

```
template<class T, int i> class Y {
  int T;            // error: template-parameter redeclared
  void f() {
    char T;         // error: template-parameter redeclared
  }
};

template<class X> class X;       // error: template-parameter redeclared
```

— *end example*]

1 In the definition of a member of a class template that appears outside of the class template definition, the name of a member of the class template hides the name of a *template-parameter* of any enclosing class templates (but not a *template-parameter* of the member if the member is a class or function template). [*Example:*

```
template<class T> struct A {
  struct B { /* ...  */ };
  typedef void C;
  void f();
  template<class U> void g(U);
};

template<class B> void A<B>::f() {
  B b;              // A's B, not the template parameter
}

template<class B> template<class C> void A<B>::g(C) {
  B b;              // A's B, not the template parameter
  C c;              // the template parameter C, not A's C
}
```

— *end example*]

2 In the definition of a member of a class template that appears outside of the namespace containing the class template definition, the name of a *template-parameter* hides the name of a member of this namespace. [*Example:*

```
namespace N {
  class C { };
  template<class T> class B {
```

```
    void f(T);
  };
}
template<class C> void N::B<C>::f(C) {
  C b;              // C is the template parameter, not N::C
}
```

— *end example*]

3

In the definition of a class template or in the definition of a member of such a template that appears outside of the template definition, for each base class which does not depend on a *template-parameter* (14.1257.1), if the name of the base class or the name of a member of the base class is the same as the name of a *template-parameter*, the base class name or member name hides the *template-parameter* name (3.875.1). [*Example:*

```
struct A {
  struct B { /* ... */ };
  int a;
  int Y;
};

template<class B, class a> struct X : A {
  B b;              // A's B
  a b;              // error: A's a isn't a type name
};
```

— *end example*]

14.1257.1 Dependent names [temp.dep]

1

Inside a template, some constructs have semantics which may differ from one instantiation to another. Such a construct *depends* on the template parameters. In particular, types and expressions may depend on the type and/or value of template parameters (as determined by the template arguments) and this determines the context for name lookup for certain names. Expressions may be *type-dependent* (on the type of a template parameter) or *value-dependent* (on the value of a non-type template parameter). In an expression of the form:

postfix-expression `(` *expression-list*$_{opt}$`)`

where the *postfix-expression* is an *unqualified-id*, the *unqualified-id* denotes a *dependent name* if

— any of the expressions in the *expression-list* is a pack expansion (14.703.1),

— any of the expressions in the *expression-list* is a type-dependent expression (14.1390.0.1), or

— if the *unqualified-id* is a *template-id* in which any of the template arguments depends on a template parameter.

If an operand of an operator is a type-dependent expression, the operator also denotes a dependent name. Such names are unbound and are looked up at the point of the template instantiation (14.1486.0.1) in both the context of the template definition and the context of the point of instantiation.

1

[*Example:*

```
template<class T> struct X : B<T> {
  typename T::A* pa;
  void f(B<T>* pb) {
    static int i = B<T>::i;
    pb->j++;
  }
};
```

the base class name `B<T>`, the type name `T::A`, the names `B<T>::i` and `pb->j` explicitly depend on the *template-parameter*. — *end example*]

1

In the definition of a class or class template, if a base class depends on a *template-parameter*, the base class scope is not examined during unqualified name lookup either at the point of definition of the class template or member or during an instantiation of the class template or member. [*Example:*

```
typedef double A;
template<class T> class B {
  typedef int A;
};
template<class T> struct X : B<T> {
  A a;              // a has type double
};
```

The type name `A` in the definition of `X<T>` binds to the typedef name defined in the global namespace scope, not to the typedef name defined in the base class `B<T>`. — *end example*] [*Example:*

```
struct A {
  struct B { /* ... */ };
  int a;
  int Y;
};

int a;

template<class T> struct Y : T {
  struct B { /* ... */ };
  B b;                          // The B defined in Y
  void f(int i) { a = i; }      // ::a
  Y* p;                         // Y<T>
};

Y<A> ya;
```

The members `A::B`, `A::a`, and `A::Y` of the template argument `A` do not affect the binding of names in `Y<A>`. — *end example*]

14.1319.0.1 Dependent types [temp.dep.type]

1

A name refers to the *current instantiation* if it is

— in the definition of a class template, a nested class of a class template, a member of a class template, or a member of a nested class of a class template, the injected-class-name (Clause 9) of the class template or nested class,

— in the definition of a primary class template or a member of a primary class template, the name of the class template followed by the template argument list of the primary template (as described below) enclosed in `<>` (or an equivalent template alias specialization),

— in the definition of a nested class of a class template, the name of the nested class referenced as a member of the current instantiation, or

— in the definition of a partial specialization or a member of a partial specialization, the name of the class template followed by the template argument list of the partial specialization enclosed in `<>` (or an equivalent template alias specialization). If the *n*th template parameter is a parameter pack, the *n*th template argument is a pack expansion (14.703.1) whose pattern is the name of the parameter pack.

1 The template argument list of a primary template is a template argument list in which the *n*th template argument has the value of the *n*th template parameter of the class template. If the *n*th template parameter is a template parameter pack (14.703.1), the *n*th template argument is a pack expansion (14.703.1) whose pattern is the name of the template parameter pack.

2 A template argument that is equivalent to a template parameter (i.e., has the same constant value or the same type as the template parameter) can be used in place of that template parameter in a reference to the current instantiation. In the case of a non-type template argument, the argument must have been given the value of the template parameter and not an expression in which the template parameter appears as a subexpression. [*Example:*

```
template <class T> class A {
  A* p1;                        // A is the current instantiation
  A<T>* p2;                     // A<T> is the current instantiation
  A<T*> p3;                     // A<T*> is not the current instantiation
  ::A<T>* p4;                   // ::A<T> is the current instantiation
  class B {
    B* p1;                      // B is the current instantiation
    A<T>::B* p2;                // A<T>::B is the current instantiation
    typename A<T*>::B* p3;      // A<T*>::B is not the
                                // current instantiation
  };
};

template <class T> class A<T*> {
  A<T*>* p1;                    // A<T*> is the current instantiation
  A<T>* p2;                     // A<T> is not the current instantiation
};

template <class T1, class T2, int I> struct B {
  B<T1, T2, I>* b1;             // refers to the current instantiation
  B<T2, T1, I>* b2;             // not the current instantiation
  typedef T1 my_T1;
  static const int my_I = I;
  static const int my_I2 = I+0;
  static const int my_I3 = my_I;
  B<my_T1, T2, my_I>* b3;       // refers to the current instantiation
  B<my_T1, T2, my_I2>* b4;      // not the current instantiation
  B<my_T1, T2, my_I3>* b5;      // refers to the current instantiation
};
```

— *end example*]

3

A name is a *member of the current instantiation* if it is

— An unqualified name that, when looked up, refers to at least one member of a class that is the current instantiation or a non-dependent base class thereof. [*Note:* This can only occur when looking up a name in a scope enclosed by the definition of a class template. — *end note*]

— A *qualified-id* in which the *nested-name-specifier* refers to the current instantiation and that, when looked up, refers to at least one member of a class that is the current instantiation or a non-dependent base class thereof. [*Note:* if no such member is found, and the current instantiation has any dependent base classes, then the *qualified-id* is a member of an unknown specialization; see below. — *end note*]

— An *id-expression* denoting the member in a class member access expression (5.1194.1) for which the type of the object expression is the current instantiation, and the *id-expression*, when looked up (3.1423.1), refers to at least one member of a class that is the current instantiation or a non-dependent base class thereof. [*Note:* if no such member is found, and the current instantiation has any dependent base classes, then the *id-expression* is a member of an unknown specialization; see below. — *end note*]

[*Example:*

```
template <class T> class A {
  static const int i = 5;
  int n1[i];         // i refers to a member of the current instantiation
  int n2[A::i];      // A::i refers to a member of the current instantiation
  int n3[A<T>::i];   // A<T>::i refers to a member of the current instantiation
  int f();
};

template <class T> int A<T>::f() {
  return i;          // i refers to a member of the current instantiation
}
```

— *end example*]

A name is a *dependent member of the current instantiation* if it is a member of the current instantiation that, when looked up, refers to at least one member of a class that is the current instantiation.

1

A name is a *member of an unknown specialization* if it is

— A *qualified-id* in which the *nested-name-specifier* names a dependent type that is not the current instantiation.

— A *qualified-id* in which the *nested-name-specifier* refers to the current instantiation, the current instantiation has at least one dependent base class, and name lookup of the *qualified-id* does not find any member of a class that is the current instantiation or a non-dependent base class thereof.

— An *id-expression* denoting the member in a class member access expression (5.1194.1) in which either

  — the type of the object expression is the current instantiation, the current instantiation has at least one dependent base class, and name lookup of the *id-expression* does not find a member of a class that is the current instantiation or a non-dependent base class thereof; or

  — the type of the object expression is dependent and is not the current instantiation.

1

If a *qualified-id* in which the *nested-name-specifier* refers to the current instantiation is not a member of the current instantiation or a member of an unknown specialization, the program is ill-formed even if the

template containing the *qualified-id* is not instantiated; no diagnostic required. Similarly, if the *id-expression* in a class member access expression for which the type of the object expression is the current instantiation does not refer to a member of the current instantiation or a member of an unknown specialization, the program is ill-formed even if the template containing the member access expression is not instantiated; no diagnostic required. [*Example:*

```
template<class T> class A {
  typedef int type;
  void f() {
    A<T>::type i;            // OK: refers to a member of the current instantiation
    typename A<T>::other j;  // error: neither a member of the current instantiation nor
                             // a member of an unknown specialization
  }
};
```

— *end example*]

1

If, for a given set of template arguments, a specialization of a template is instantiated that refers to a member of the current instantiation with a *qualified-id* or class member access expression, the name in the *qualified-id* or class member access expression is looked up in the template instantiation context. If the result of this lookup differs from the result of name lookup in the template definition context, name lookup is ambiguous. [*Note:* the result of name lookup differs only when the member of the current instantiation was found in a non-dependent base class of the current instantiation and a member with the same name is also introduced by the substitution for a dependent base class of the current instantiation. — *end note*]

1

A type is dependent if it is

— a template parameter,

— a member of an unknown specialization,

— a nested class or enumeration that is a dependent member of the current instantiation,

— a cv-qualified type where the cv-unqualified type is dependent,

— a compound type constructed from any dependent type,

— an array type constructed from any dependent type or whose size is specified by a constant expression that is value-dependent,

— a *simple-template-id* in which either the template name is a template parameter or any of the template arguments is a dependent type or an expression that is type-dependent or value-dependent, or

— denoted by `decltype(`*expression*`)`, where *expression* is type-dependent (14.1390.0.1).

1 [*Note:* Because typedefs do not introduce new types, but instead simply refer to other types, a name that refers to a typedef that is a member of the current instantiation is dependent only if the type referred to is dependent. — *end note*]

14.1390.0.1 Type-dependent expressions [temp.dep.expr]

1

Except as described below, an expression is type-dependent if any subexpression is type-dependent.

1

`this` is type-dependent if the class type of the enclosing member function is dependent (14.1319.0.1).

1

An *id-expression* is type-dependent if it contains

— an *identifier* associated by name lookup with one or more declarations declared with a dependent type,

— an *identifier* associated by name lookup with one or more declarations of member functions of the current instantiation declared with a return type that contains a placeholder type (7.832.0.1),

— a *template-id* that is dependent,

— a *conversion-function-id* that specifies a dependent type, or

— a *nested-name-specifier* or a *qualified-id* that names a member of an unknown specialization;

or if it names a dependent member of the current instantiation that is a static data member of type "array of unknown bound of `T`" for some `T` (14.634.0.1). Expressions of the following forms are type-dependent only if the type specified by the *type-id*, *simple-type-specifier* or *new-type-id* is dependent, even if any subexpression is type-dependent:

simple-type-specifier `(` *expression-list*$_{opt}$`)`
`::`$_{opt}$ `new` *new-placement*$_{opt}$ *new-type-id* *new-initializer*$_{opt}$
`::`$_{opt}$ `new` *new-placement*$_{opt}$`(` *type-id* `)` *new-initializer*$_{opt}$
`dynamic_cast <` *type-id* `> (` *expression* `)`
`static_cast <` *type-id* `> (` *expression* `)`
`const_cast <` *type-id* `> (` *expression* `)`
`reinterpret_cast <` *type-id* `> (` *expression* `)`
`(` *type-id* `)` *cast-expression*

1

Expressions of the following forms are never type-dependent (because the type of the expression cannot be dependent):

literal
postfix-expression `.` *pseudo-destructor-name*
postfix-expression `->` *pseudo-destructor-name*
`sizeof` *unary-expression*
`sizeof (` *type-id* `)`
`sizeof ... (` *identifier* `)`
`alignof (` *type-id* `)`
`typeid (` *expression* `)`
`typeid (` *type-id* `)`
`::`$_{opt}$ `delete` *cast-expression*
`::`$_{opt}$ `delete [ ]` *cast-expression*
`throw` *assignment-expression*$_{opt}$
`noexcept (` *expression* `)`

[*Note:* For the standard library macro `offsetof`, see 18.2. — *end note*]

1

A class member access expression (5.1194.1) is type-dependent if the expression refers to a member of the current instantiation and the type of the referenced member is dependent, or the class member access expression refers to a member of an unknown specialization. [*Note:* In an expression of the form `x.y` or `xp->y` the type of the expression is usually the type of the member `y` of the class of `x` (or the class pointed to by `xp`). However, if `x` or `xp` refers to a dependent type that is not the current instantiation, the type of

`y` is always dependent. If `x` or `xp` refers to a non-dependent type or refers to the current instantiation, the type of `y` is the type of the class member access expression. — *end note*]

14.1430.0.1 Value-dependent expressions [temp.dep.constexpr]

1

Except as described below, a constant expression is value-dependent if any subexpression is value-dependent.

1

An *id-expression* is value-dependent if:

— it is a name declared with a dependent type,

— it is the name of a non-type template parameter,

— it names a member of an unknown specialization,

— it names a static data member that is a dependent member of the current instantiation and is not initialized in a *member-declarator*,

— it names a static member function that is a dependent member of the current instantiation, or

— it is a constant with literal type and is initialized with an expression that is value-dependent.

Expressions of the following form are value-dependent if the *unary-expression* or *expression* is type-dependent or the *type-id* is dependent:

`sizeof` *unary-expression*
`sizeof (` *type-id* `)`
`typeid (` *expression* `)`
`typeid (` *type-id* `)`
`alignof (` *type-id* `)`
`noexcept (` *expression* `)`

[*Note:* For the standard library macro `offsetof`, see 18.2. — *end note*]

1 Expressions of the following form are value-dependent if either the *type-id* or *simple-type-specifier* is dependent or the *expression* or *cast-expression* is value-dependent:

simple-type-specifier `(` *expression-list*$_{opt}$ `)`
`static_cast <` *type-id* `> (` *expression* `)`
`const_cast <` *type-id* `> (` *expression* `)`
`reinterpret_cast <` *type-id* `> (` *expression* `)`
`(` *type-id* `)` *cast-expression*

2 Expressions of the following form are value-dependent:

`sizeof ... (` *identifier* `)`

3

An expression of the form `&`*qualified-id* where the *qualified-id* names a dependent member of the current instantiation is value-dependent.

14.1439.0.1 Dependent template arguments [temp.dep.temp]

1

A type *template-argument* is dependent if the type it specifies is dependent.

1

A non-type *template-argument* is dependent if its type is dependent or the constant expression it specifies is value-dependent.

1 Furthermore, a non-type *template-argument* is dependent if the corresponding non-type *template-parameter* is of reference or pointer type and the *template-argument* designates or points to a member of the current instantiation or a member of a dependent type.

2 A template *template-argument* is dependent if it names a *template-parameter* or is a *qualified-id* that refers to a member of an unknown specialization.

14.1444.1 Non-dependent names [temp.nondep]

1

Non-dependent names used in a template definition are found using the usual name lookup and bound at the point they are used. [*Example:*

```
void g(double);
void h();

template<class T> class Z {
public:
  void f() {
    g(1);           // calls g(double)
    h++;            // ill-formed: cannot increment function;
                    // this could be diagnosed either here or
                    // at the point of instantiation
  }
};

void g(int);        // not in scope at the point of the template
                    // definition, not considered for the call g(1)
```

— *end example*]

14.1452.1 Dependent name resolution [temp.dep.res]

1

In resolving dependent names, names from the following sources are considered:

— Declarations that are visible at the point of definition of the template.

— Declarations from namespaces associated with the types of the function arguments both from the instantiation context (14.1486.0.1) and from the definition context.

14.1486.0.1 Point of instantiation [temp.point]

1

For a function template specialization, a member function template specialization, or a specialization for a member function or static data member of a class template, if the specialization is implicitly instantiated because it is referenced from within another template specialization and the context from which it is referenced depends on a template parameter, the point of instantiation of the specialization is the point of instantiation of the enclosing specialization. Otherwise, the point of instantiation for such a specialization immediately follows the namespace scope declaration or definition that refers to the specialization.

1

If a function template or member function of a class template is called in a way which uses the definition of a default argument of that function template or member function, the point of instantiation of the default argument is the point of instantiation of the function template or member function specialization.

1 For an *exception-specification* of a function template specialization or specialization of a member function of a class template, if the *exception-specification* is implicitly instantiated because it is needed by another template specialization and the context that requires it depends on a template parameter, the point of instantiation of the *exception-specification* is the point of instantiation of the specialization that requires it. Otherwise, the point of instantiation for such an *exception-specification* immediately follows the namespace scope declaration or definition that requires the *exception-specification*.

2

For a class template specialization, a class member template specialization, or a specialization for a class member of a class template, if the specialization is implicitly instantiated because it is referenced from within another template specialization, if the context from which the specialization is referenced depends on a template parameter, and if the specialization is not instantiated previous to the instantiation of the enclosing template, the point of instantiation is immediately before the point of instantiation of the enclosing template. Otherwise, the point of instantiation for such a specialization immediately precedes the namespace scope declaration or definition that refers to the specialization.

1

If a virtual function is implicitly instantiated, its point of instantiation is immediately following the point of instantiation of its enclosing class template specialization.

1

An explicit instantiation definition is an instantiation point for the specialization or specializations specified by the explicit instantiation.

1

The instantiation context of an expression that depends on the template arguments is the set of declarations with external linkage declared prior to the point of instantiation of the template specialization in the same translation unit.

1

A specialization for a function template, a member function template, or of a member function or static data member of a class template may have multiple points of instantiations within a translation unit, and in addition to the points of instantiation described above, for any such specialization that has a point of instantiation within the translation unit, the end of the translation unit is also considered a point of instantiation. A specialization for a class template has at most one point of instantiation within a translation unit. A specialization for any template may have points of instantiation in multiple translation units. If two different points of instantiation give a template specialization different meanings according to the one definition rule (3.91), the program is ill-formed, no diagnostic required.

14.1532.0.1 Candidate functions [temp.dep.candidate]

1

For a function call where the *postfix-expression* is a dependent name, the candidate functions are found using the usual lookup rules (3.952.1, 3.1092.1) except that:

— For the part of the lookup using unqualified name lookup (3.952.1), only function declarations from the template definition context are found.

— For the part of the lookup using associated namespaces (3.1092.1), only function declarations found in either the template definition context or the template instantiation context are found.

If the call would be ill-formed or would find a better match had the lookup within the associated namespaces considered all the function declarations with external linkage introduced in those namespaces in all translation units, not just considering those declarations found in the template definition and template instantiation contexts, then the program has undefined behavior.

14.1608.1 Friend names declared within a class template [temp.inject]

1

Friend classes or functions can be declared within a class template. When a template is instantiated, the names of its friends are treated as if the specialization had been explicitly declared at its point of instantiation.

1

As with non-template classes, the names of namespace-scope friend functions of a class template specialization are not visible during an ordinary lookup unless explicitly declared at namespace scope (11.94). Such names may be found under the rules for associated classes (3.1092.1).[142] [*Example:*

```
template<typename T> struct number {
  number(int);
  friend number gcd(number x, number y) { return 0; };
};

void g() {
  number<double> a(3), b(4);
  a = gcd(a,b);      // finds gcd because number<double> is an
                     // associated class, making gcd visible
                     // in its namespace (global scope)
  b = gcd(3,4);      // ill-formed; gcd is not visible
}
```

— *end example*]

14.1614 Template instantiation and specialization [temp.spec]

1

The act of instantiating a function, a class, a member of a class template or a member template is referred to as *template instantiation*.

1

A function instantiated from a function template is called an instantiated function. A class instantiated from a class template is called an instantiated class. A member function, a member class, a member enumeration, or a static data member of a class template instantiated from the member definition of the class template is called, respectively, an instantiated member function, member class, member enumeration, or static data member. A member function instantiated from a member function template is called an instantiated member function. A member class instantiated from a member class template is called an instantiated member class.

1

An explicit specialization may be declared for a function template, a class template, a member of a class template or a member template. An explicit specialization declaration is introduced by `template<>`. In an explicit specialization declaration for a class template, a member of a class template or a class member template, the name of the class that is explicitly specialized shall be a *simple-template-id*. In the explicit specialization declaration for a function template or a member function template, the name of the function or member function explicitly specialized may be a *template-id*. [*Example:*

```
template<class T = int> struct A {
  static int x;
};
template<class U> void g(U) { }
```

142) Friend declarations do not introduce new names into any scope, either when the template is declared or when it is instantiated.

```
template<> struct A<double> { };        // specialize for T == double
template<> struct A<> { };              // specialize for T == int
template<> void g(char) { }             // specialize for U == char
                                        // U is deduced from the parameter type
template<> void g<int>(int) { }         // specialize for U == int
template<> int A<char>::x = 0;          // specialize for T == char

template<class T = int> struct B {
  static int x;
};
template<> int B<>::x = 1;              // specialize for T == int
```

— *end example*]

1

An instantiated template specialization can be either implicitly instantiated (14.1644.1) for a given argument list or be explicitly instantiated (14.1841.1). A specialization is a class, function, or class member that is either instantiated or explicitly specialized (14.1921.1).

1

For a given template and a given set of *template-arguments*,

— an explicit instantiation definition shall appear at most once in a program,

— an explicit specialization shall be defined at most once in a program (according to 3.91), and

— both an explicit instantiation and a declaration of an explicit specialization shall not appear in a program unless the explicit instantiation follows a declaration of the explicit specialization.

An implementation is not required to diagnose a violation of this rule.

1

Each class template specialization instantiated from a template has its own copy of any static members. [*Example:*

```
template<class T> class X {
  static T s;
};
template<class T> T X<T>::s = 0;
X<int> aa;
X<char*> bb;
```

`X<int>` has a static member `s` of type `int` and `X<char*>` has a static member `s` of type `char*`. — *end example*]

14.1644.1 Implicit instantiation [temp.inst]

1

Unless a class template specialization has been explicitly instantiated (14.1841.1) or explicitly specialized (14.1921.1), the class template specialization is implicitly instantiated when the specialization is referenced in a context that requires a completely-defined object type or when the completeness of the class type affects the semantics of the program. The implicit instantiation of a class template specialization causes the implicit instantiation of the declarations, but not of the definitions, default arguments, or *exception-specifications* of the class member functions, member classes, scoped member enumerations, static data members and member templates; and it causes the implicit instantiation of the definitions of unscoped

member enumerations and member anonymous unions. However, for the purpose of determining whether an instantiated redeclaration of a member is valid according to 9.194, a declaration that corresponds to a definition in the template is considered to be a definition. [*Example:*

```
template<class T, class U>
struct Outer {
  template<class X, class Y> struct Inner;
  template<class Y> struct Inner<T, Y>;          // #1a
  template<class Y> struct Inner<T, Y> { };      // #1b; OK: valid redeclaration of #1a
  template<class Y> struct Inner<U, Y> { };      // #2
};

Outer<int, int> outer;                           // error at #2
```

`Outer<int, int>::Inner<int, Y>` is redeclared at #1b. (It is not defined but noted as being associated with a definition in `Outer<T, U>`.) #2 is also a redeclaration of #1a. It is noted as associated with a definition, so it is an invalid redeclaration of the same partial specialization. — *end example*]

1

Unless a member of a class template or a member template has been explicitly instantiated or explicitly specialized, the specialization of the member is implicitly instantiated when the specialization is referenced in a context that requires the member definition to exist; in particular, the initialization (and any associated side-effects) of a static data member does not occur unless the static data member is itself used in a way that requires the definition of the static data member to exist.

1

Unless a function template specialization has been explicitly instantiated or explicitly specialized, the function template specialization is implicitly instantiated when the specialization is referenced in a context that requires a function definition to exist. Unless a call is to a function template explicit specialization or to a member function of an explicitly specialized class template, a default argument for a function template or a member function of a class template is implicitly instantiated when the function is called in a context that requires the value of the default argument.

1

[*Example:*

```
template<class T> struct Z {
  void f();
  void g();
};

void h() {
  Z<int> a;          // instantiation of class Z<int> required
  Z<char>* p;        // instantiation of class Z<char> not required
  Z<double>* q;      // instantiation of class Z<double> not required

  a.f();             // instantiation of Z<int>::f() required
  p->g();            // instantiation of class Z<char> required, and
                     // instantiation of Z<char>::g() required
}
```

Nothing in this example requires `class Z<double>`, `Z<int>::g()`, or `Z<char>::f()` to be implicitly instantiated. — *end example*]

1 Unless a variable template specialization has been explicitly instantiated or explicitly specialized, the variable template specialization is implicitly instantiated when the specialization is used. A default template

argument for a variable template is implicitly instantiated when the variable template is referenced in a context that requires the value of the default argument.

2

A class template specialization is implicitly instantiated if the class type is used in a context that requires a completely-defined object type or if the completeness of the class type might affect the semantics of the program. [*Note:* In particular, if the semantics of an expression depend on the member or base class lists of a class template specialization, the class template specialization is implicitly generated. For instance, deleting a pointer to class type depends on whether or not the class declares a destructor, and conversion between pointer to class types depends on the inheritance relationship between the two classes involved. — *end note*] [*Example:*

```
template<class T> class B { /* ... */ };
template<class T> class D : public B<T> { /* ... */ };

void f(void*);
void f(B<int>*);

void g(D<int>* p, D<char>* pp, D<double>* ppp) {
  f(p);              // instantiation of D<int> required: call f(B<int>*)
  B<char>* q = pp;   // instantiation of D<char> required:
                     // convert D<char>* to B<char>*
  delete ppp;        // instantiation of D<double> required
}
```

— *end example*]

1

If the overload resolution process can determine the correct function to call without instantiating a class template definition, it is unspecified whether that instantiation actually takes place. [*Example:*

```
template <class T> struct S {
  operator int();
};

void f(int);
void f(S<int>&);
void f(S<float>);

void g(S<int>& sr) {
  f(sr);             // instantiation of S<int> allowed but not required
                     // instantiation of S<float> allowed but not required
};
```

— *end example*]

1

If an implicit instantiation of a class template specialization is required and the template is declared but not defined, the program is ill-formed. [*Example:*

```
template<class T> class X;

X<char> ch;          // error: definition of X required
```

— *end example*]

1

The implicit instantiation of a class template does not cause any static data members of that class to be implicitly instantiated.

1

If a function template or a member function template specialization is used in a way that involves overload resolution, a declaration of the specialization is implicitly instantiated (14.2662.1).

1

An implementation shall not implicitly instantiate a function template, a variable template, a member template, a non-virtual member function, a member class, or a static data member of a class template that does not require instantiation. It is unspecified whether or not an implementation implicitly instantiates a virtual member function of a class template if the virtual member function would not otherwise be instantiated. The use of a template specialization in a default argument shall not cause the template to be implicitly instantiated except that a class template may be instantiated where its complete type is needed to determine the correctness of the default argument. The use of a default argument in a function call causes specializations in the default argument to be implicitly instantiated.

1

Implicitly instantiated class, function, and variable template specializations are placed in the namespace where the template is defined. Implicitly instantiated specializations for members of a class template are placed in the namespace where the enclosing class template is defined. Implicitly instantiated member templates are placed in the namespace where the enclosing class or class template is defined. [*Example:*

```
namespace N {
  template<class T> class List {
  public:
    T* get();
  };
}

template<class K, class V> class Map {
public:
  N::List<V> lt;
  V get(K);
};

void g(Map<const char*,int>& m) {
  int i = m.get("Nicholas");
}
```

a call of `lt.get()` from `Map<const char*,int>::get()` would place `List<int>::get()` in the namespace `N` rather than in the global namespace. — *end example*]

1

If a function template `f` is called in a way that requires a default argument to be used, the dependent names are looked up, the semantics constraints are checked, and the instantiation of any template used in the default argument is done as if the default argument had been an initializer used in a function template specialization with the same scope, the same template parameters and the same access as that of the function template `f` used at that point, except that the scope in which a closure type is declared (5.509.1) – and therefore its associated namespaces – remain as determined from the context of the definition for the default argument. This analysis is called *default argument instantiation*. The instantiated default argument is then used as the argument of `f`.

1

Each default argument is instantiated independently. [*Example:*

```
template<class T> void f(T x, T y = ydef(T()), T z = zdef(T()));

class  A { };

A zdef(A);

void g(A a, A b, A c) {
  f(a, b, c);        // no default argument instantiation
  f(a, b);           // default argument z = zdef(T()) instantiated
  f(a);              // ill-formed; ydef is not declared
}
```

— *end example*]

1 The *exception-specification* of a function template specialization is not instantiated along with the function declaration; it is instantiated when needed (15.228). If such an *exception-specification* is needed but has not yet been instantiated, the dependent names are looked up, the semantics constraints are checked, and the instantiation of any template used in the *exception-specification* is done as if it were being done as part of instantiating the declaration of the specialization at that point.

2 [*Note:* 14.1486.0.1 defines the point of instantiation of a template specialization. — *end note*]

3

There is an implementation-defined quantity that specifies the limit on the total depth of recursive instantiations, which could involve more than one template. The result of an infinite recursion in instantiation is undefined. [*Example:*

```
template<class T> class X {
  X<T>* p;           // OK
  X<T*> a;           // implicit generation of X<T> requires
                     // the implicit instantiation of X<T*> which requires
                     // the implicit instantiation of X<T**> which ...
};
```

— *end example*]

14.1841.1 Explicit instantiation [temp.explicit]

1

A class, function, variable, or member template specialization can be explicitly instantiated from its template. A member function, member class or static data member of a class template can be explicitly instantiated from the member definition associated with its class template. An explicit instantiation of a function template or member function of a class template shall not use the `inline` or `constexpr` specifiers.

1

The syntax for explicit instantiation is:

explicit-instantiation:
  `extern`$_{opt}$ `template` *declaration*

There are two forms of explicit instantiation: an explicit instantiation definition and an explicit instantiation declaration. An explicit instantiation declaration begins with the `extern` keyword.

1

If the explicit instantiation is for a class or member class, the *elaborated-type-specifier* in the *declaration* shall include a *simple-template-id*. If the explicit instantiation is for a function or member function, the *unqualified-*

id in the *declaration* shall be either a *template-id* or, where all template arguments can be deduced, a *template-name* or *operator-function-id*. [*Note:* The declaration may declare a *qualified-id*, in which case the *unqualified-id* of the *qualified-id* must be a *template-id*. — *end note*] If the explicit instantiation is for a member function, a member class or a static data member of a class template specialization, the name of the class template specialization in the *qualified-id* for the member name shall be a *simple-template-id*. If the explicit instantiation is for a variable, the *unqualified-id* in the declaration shall be a *template-id*. An explicit instantiation shall appear in an enclosing namespace of its template. If the name declared in the explicit instantiation is an unqualified name, the explicit instantiation shall appear in the namespace where its template is declared or, if that namespace is inline (7.1149.1), any namespace from its enclosing namespace set. [*Note:* Regarding qualified names in declarators, see 8.56. — *end note*] [*Example:*

```
template<class T> class Array { void mf(); };
template class Array<char>;
template void Array<int>::mf();

template<class T> void sort(Array<T>& v) { /* ... */ }
template void sort(Array<char>&);       // argument is deduced here

namespace N {
  template<class T> void f(T&) { }
}
template void N::f<int>(int&);
```

— *end example*]

1

A declaration of a function template, a variable template, a member function or static data member of a class template, or a member function template of a class or class template shall precede an explicit instantiation of that entity. A definition of a class template, a member class of a class template, or a member class template of a class or class template shall precede an explicit instantiation of that entity unless the explicit instantiation is preceded by an explicit specialization of the entity with the same template arguments. If the *declaration* of the explicit instantiation names an implicitly-declared special member function (Clause 12), the program is ill-formed.

1

For a given set of template arguments, if an explicit instantiation of a template appears after a declaration of an explicit specialization for that template, the explicit instantiation has no effect. Otherwise, for an explicit instantiation definition the definition of a function template, a variable template, a member function template, or a member function or static data member of a class template shall be present in every translation unit in which it is explicitly instantiated.

1

An explicit instantiation of a class, function template, or variable template specialization is placed in the namespace in which the template is defined. An explicit instantiation for a member of a class template is placed in the namespace where the enclosing class template is defined. An explicit instantiation for a member template is placed in the namespace where the enclosing class or class template is defined. [*Example:*

```
namespace N {
  template<class T> class Y { void mf() { } };
}

template class Y<int>;                  // error: class template Y not visible
                                        // in the global namespace

using N::Y;
```

```
template class Y<int>;                    // error: explicit instantiation outside of the
                                          // namespace of the template

template class N::Y<char*>;               // OK: explicit instantiation in namespace N
template void N::Y<double>::mf();         // OK: explicit instantiation
                                          // in namespace N
```

— *end example*]

1 A trailing *template-argument* can be left unspecified in an explicit instantiation of a function template specialization or of a member function template specialization provided it can be deduced from the type of a function parameter (14.2114.1). [*Example:*

```
template<class T> class Array { /* ... */ };
template<class T> void sort(Array<T>& v) { /* ... */ }

// instantiate sort(Array<int>&) - template-argument deduced
template void sort<>(Array<int>&);
```

— *end example*]

2

An explicit instantiation that names a class template specialization is also an explicit instantiation of the same kind (declaration or definition) of each of its members (not including members inherited from base classes and members that are templates) that has not been previously explicitly specialized in the translation unit containing the explicit instantiation, except as described below. [*Note:* In addition, it will typically be an explicit instantiation of certain implementation-dependent data about the class. — *end note*]

1

An explicit instantiation definition that names a class template specialization explicitly instantiates the class template specialization and is an explicit instantiation definition of only those members that have been defined at the point of instantiation.

1

Except for inline functions, declarations with types deduced from their initializer or return value (7.832.0.1), `const` variables of literal types, variables of reference types, and class template specializations, explicit instantiation declarations have the effect of suppressing the implicit instantiation of the entity to which they refer. [*Note:* The intent is that an inline function that is the subject of an explicit instantiation declaration will still be implicitly instantiated when odr-used (3.91) so that the body can be considered for inlining, but that no out-of-line copy of the inline function would be generated in the translation unit. — *end note*]

1

If an entity is the subject of both an explicit instantiation declaration and an explicit instantiation definition in the same translation unit, the definition shall follow the declaration. An entity that is the subject of an explicit instantiation declaration and that is also used in a way that would otherwise cause an implicit instantiation (14.1644.1) in the translation unit shall be the subject of an explicit instantiation definition somewhere in the program; otherwise the program is ill-formed, no diagnostic required. [*Note:* This rule does apply to inline functions even though an explicit instantiation declaration of such an entity has no other normative effect. This is needed to ensure that if the address of an inline function is taken in a translation unit in which the implementation chose to suppress the out-of-line body, another translation unit will supply the body. — *end note*] An explicit instantiation declaration shall not name a specialization of a template with internal linkage.

1

The usual access checking rules do not apply to names used to specify explicit instantiations. [*Note:* In particular, the template arguments and names used in the function declarator (including parameter types, return types and exception specifications) may be private types or objects which would normally not be accessible and the template may be a member template or member function which would not normally be accessible. — *end note*]

1 An explicit instantiation does not constitute a use of a default argument, so default argument instantiation is not done. [*Example:*

```
char* p = 0;
template<class T> T g(T x = &p) { return x; }
template int g<int>(int);       // OK even though &p isn't an int.
```

— *end example*]

14.1921.1 Explicit specialization [temp.expl.spec]

1

An explicit specialization of any of the following:

— function template

— class template

— variable template

— member function of a class template

— static data member of a class template

— member class of a class template

— member enumeration of a class template

— member class template of a class or class template

— member function template of a class or class template

can be declared by a declaration introduced by `template<>`; that is:

explicit-specialization:
`template < >` *declaration*

[*Example:*

```
template<class T> class stream;

template<> class stream<char> { /* ... */ };

template<class T> class Array { /* ... */ };
template<class T> void sort(Array<T>& v) { /* ... */ }

template<> void sort<char*>(Array<char*>&) ;
```

Given these declarations, `stream<char>` will be used as the definition of streams of `char`s; other streams will be handled by class template specializations instantiated from the class template. Similarly, `sort<char*>` will be used as the sort function for arguments of type `Array<char*>`; other `Array` types will be sorted by functions generated from the template. — *end example*]

1

An explicit specialization shall be declared in a namespace enclosing the specialized template. An explicit specialization whose *declarator-id* is not qualified shall be declared in the nearest enclosing namespace of the template, or, if the namespace is inline (7.1149.1), any namespace from its enclosing namespace set. Such a declaration may also be a definition. If the declaration is not a definition, the specialization may be defined later (7.1215.0.1).

1

A declaration of a function template, class template, or variable template being explicitly specialized shall precede the declaration of the explicit specialization. [*Note:* A declaration, but not a definition of the template is required. — *end note*] The definition of a class or class template shall precede the declaration of an explicit specialization for a member template of the class or class template. [*Example:*

```
template<> class X<int> { /* ... */ };          // error: X not a template

template<class T> class X;

template<> class X<char*> { /* ... */ };        // OK: X is a template
```

— *end example*]

1

A member function, a member function template, a member class, a member enumeration, a member class template, a static data member, or a static data member template of a class template may be explicitly specialized for a class specialization that is implicitly instantiated; in this case, the definition of the class template shall precede the explicit specialization for the member of the class template. If such an explicit specialization for the member of a class template names an implicitly-declared special member function (Clause 12), the program is ill-formed.

1

A member of an explicitly specialized class is not implicitly instantiated from the member declaration of the class template; instead, the member of the class template specialization shall itself be explicitly defined if its definition is required. In this case, the definition of the class template explicit specialization shall be in scope at the point at which the member is defined. The definition of an explicitly specialized class is unrelated to the definition of a generated specialization. That is, its members need not have the same names, types, etc. as the members of a generated specialization. Members of an explicitly specialized class template are defined in the same manner as members of normal classes, and not using the `template<>` syntax. The same is true when defining a member of an explicitly specialized member class. However, `template<>` is used in defining a member of an explicitly specialized member class template that is specialized as a class template. [*Example:*

```
template<class T> struct A {
  struct B { };
  template<class U> struct C { };
};

template<> struct A<int> {
  void f(int);
};

void h() {
  A<int> a;
  a.f(16);          // A<int>::f must be defined somewhere
}

// template<> not used for a member of an
```

```
// explicitly specialized class template
void A<int>::f(int) { /* ... */ }

template<> struct A<char>::B {
  void f();
};
// template<> also not used when defining a member of
// an explicitly specialized member class
void A<char>::B::f() { /* ... */ }

template<> template<class U> struct A<char>::C {
  void f();
};
// template<> is used when defining a member of an explicitly
// specialized member class template specialized as a class template
template<>
template<class U> void A<char>::C<U>::f() { /* ... */ }

template<> struct A<short>::B {
  void f();
};
template<> void A<short>::B::f() { /* ... */ }  // error: template<> not permitted

template<> template<class U> struct A<short>::C {
  void f();
};
template<class U> void A<short>::C<U>::f() { /* ... */ }  // error: template<> required
```

— *end example*]

1

If a template, a member template or a member of a class template is explicitly specialized then that specialization shall be declared before the first use of that specialization that would cause an implicit instantiation to take place, in every translation unit in which such a use occurs; no diagnostic is required. If the program does not provide a definition for an explicit specialization and either the specialization is used in a way that would cause an implicit instantiation to take place or the member is a virtual member function, the program is ill-formed, no diagnostic required. An implicit instantiation is never generated for an explicit specialization that is declared but not defined. [*Example:*

```
class String { };
template<class T> class Array { /* ... */ };
template<class T> void sort(Array<T>& v) { /* ... */ }

void f(Array<String>& v) {
  sort(v);          // use primary template
                    // sort(Array<T>&), T is String
}

template<> void sort<String>(Array<String>& v); // error: specialization
                                                // after use of primary template
template<> void sort<>(Array<char*>& v);        // OK: sort<char*> not yet used
template<class T> struct A {
  enum E : T;
  enum class S : T;
};
```

```
template<> enum A<int>::E : int { eint };          // OK
template<> enum class A<int>::S : int { sint };    // OK
template<class T> enum A<T>::E : T { eT };
template<class T> enum class A<T>::S : T { sT };
template<> enum A<char>::E : char { echar };        // ill-formed, A<char>::E was instantiated
                                                    // when A<char> was instantiated
template<> enum class A<char>::S : char { schar }; // OK
```

— *end example*]

1

The placement of explicit specialization declarations for function templates, class templates, variable templates, member functions of class templates, static data members of class templates, member classes of class templates, member enumerations of class templates, member class templates of class templates, member function templates of class templates, static data member templates of class templates, member functions of member templates of class templates, member functions of member templates of non-template classes, static data member templates of non-template classes, member function templates of member classes of class templates, etc., and the placement of partial specialization declarations of class templates, variable templates, member class templates of non-template classes, static data member templates of non-template classes, member class templates of class templates, etc., can affect whether a program is well-formed according to the relative positioning of the explicit specialization declarations and their points of instantiation in the translation unit as specified above and below. When writing a specialization, be careful about its location; or to make it compile will be such a trial as to kindle its self-immolation.

1

A template explicit specialization is in the scope of the namespace in which the template was defined. [*Example:*

```
namespace N {
  template<class T> class X { /* ... */ };
  template<class T> class Y { /* ... */ };

  template<> class X<int> { /* ... */ };          // OK: specialization
                                                  // in same namespace
  template<> class Y<double>;                     // forward declare intent to
                                                  // specialize for double
}

template<> class N::Y<double> { /* ... */ };      // OK: specialization
                                                  // in enclosing namespace
template<> class N::Y<short> { /* ... */ };       // OK: specialization
                                                  // in enclosing namespace
```

— *end example*]

1 A *simple-template-id* that names a class template explicit specialization that has been declared but not defined can be used exactly like the names of other incompletely-defined classes (3.2955). [*Example:*

```
template<class T> class X;      // X is a class template
template<> class X<int>;

X<int>* p;                      // OK: pointer to declared class X<int>
X<int> x;                       // error: object of incomplete class X<int>
```

— *end example*]

2

A trailing *template-argument* can be left unspecified in the *template-id* naming an explicit function template specialization provided it can be deduced from the function argument type. [*Example:*

```
template<class T> class Array { /* ... */ };
template<class T> void sort(Array<T>& v);

// explicit specialization for sort(Array<int>&)
// with deduced template-argument of type int
template<> void sort(Array<int>&);
```

— *end example*]

1 A function with the same name as a template and a type that exactly matches that of a template specialization is not an explicit specialization (14.972.1).

2

An explicit specialization of a function template is inline only if it is declared with the `inline` specifier or defined as deleted, and independently of whether its function template is inline. [*Example:*

```
template<class T> void f(T) { /* ... */ }
template<class T> inline T g(T) { /* ... */ }

template<> inline void f<>(int) { /* ... */ }    // OK: inline
template<> int g<>(int) { /* ... */ }            // OK: not inline
```

— *end example*]

1

An explicit specialization of a static data member of a template or an explicit specialization of a static data member template is a definition if the declaration includes an initializer; otherwise, it is a declaration. [*Note:* The definition of a static data member of a template that requires default initialization must use a *braced-init-list*:

```
template<> X Q<int>::x;          // declaration
template<> X Q<int>::x ();       // error: declares a function
template<> X Q<int>::x { };      // definition
```

— *end note*]

1

A member or a member template of a class template may be explicitly specialized for a given implicit instantiation of the class template, even if the member or member template is defined in the class template definition. An explicit specialization of a member or member template is specified using the syntax for explicit specialization. [*Example:*

```
template<class T> struct A {
  void f(T);
  template<class X1> void g1(T, X1);
  template<class X2> void g2(T, X2);
  void h(T) { }
};

// specialization
template<> void A<int>::f(int);

// out of class member template definition
template<class T> template<class X1> void A<T>::g1(T, X1) { }
```

```
// member template specialization
template<> template<class X1> void A<int>::g1(int, X1);

//member template specialization
template<> template<>
  void A<int>::g1(int, char);           // X1 deduced as char
template<> template<>
  void A<int>::g2<char>(int, char);     // X2 specified as char

// member specialization even if defined in class definition
template<> void A<int>::h(int) { }
```

— *end example*]

1

A member or a member template may be nested within many enclosing class templates. In an explicit specialization for such a member, the member declaration shall be preceded by a `template<>` for each enclosing class template that is explicitly specialized. [*Example:*

```
template<class T1> class A {
  template<class T2> class B {
    void mf();
  };
};
template<> template<> class A<int>::B<double>;
template<> template<> void A<char>::B<char>::mf();
```

— *end example*]

1

In an explicit specialization declaration for a member of a class template or a member template that appears in namespace scope, the member template and some of its enclosing class templates may remain unspecialized, except that the declaration shall not explicitly specialize a class member template if its enclosing class templates are not explicitly specialized as well. In such explicit specialization declaration, the keyword `template` followed by a *template-parameter-list* shall be provided instead of the `template<>` preceding the explicit specialization declaration of the member. The types of the *template-parameters* in the *template-parameter-list* shall be the same as those specified in the primary template definition. [*Example:*

```
template <class T1> class A {
  template<class T2> class B {
    template<class T3> void mf1(T3);
    void mf2();
  };
};
template <> template <class X>
  class A<int>::B {
      template <class T> void mf1(T);
  };
template <> template <> template<class T>
  void A<int>::B<double>::mf1(T t) { }
template <class Y> template <>
  void A<Y>::B<double>::mf2() { }       // ill-formed; B<double> is specialized but
                                        // its enclosing class template A is not
```

— *end example*]

1 A specialization of a member function template, member class template, or static data member template of a non-specialized class template is itself a template.

2 An explicit specialization declaration shall not be a friend declaration.

3 Default function arguments shall not be specified in a declaration or a definition for one of the following explicit specializations:

(3.1) — the explicit specialization of a function template;

(3.2) — the explicit specialization of a member function template;

(3.3) — the explicit specialization of a member function of a class template where the class template specialization to which the member function specialization belongs is implicitly instantiated. [*Note:* Default function arguments may be specified in the declaration or definition of a member function of a class template specialization that is explicitly specialized. — *end note*]

14.2062 Function template specializations [temp.fct.spec]

1

A function instantiated from a function template is called a function template specialization; so is an explicit specialization of a function template. Template arguments can be explicitly specified when naming the function template specialization, deduced from the context (e.g., deduced from the function arguments in a call to the function template specialization, see 14.2114.1), or obtained from default template arguments.

1

Each function template specialization instantiated from a template has its own copy of any static variable. [*Example:*

```
template<class T> void f(T* p) {
  static T s;
};

void g(int a, char* b) {
  f(&a);            // calls f<int>(int*)
  f(&b);            // calls f<char*>(char**)
}
```

Here `f<int>(int*)` has a static variable `s` of type `int` and `f<char*>(char**)` has a static variable `s` of type `char*`. — *end example*]

14.2075.1 Explicit template argument specification [temp.arg.explicit]

1

Template arguments can be specified when referring to a function template specialization by qualifying the function template name with the list of *template-arguments* in the same way as *template-arguments* are specified in uses of a class template specialization. [*Example:*

```
template<class T> void sort(Array<T>& v);
void f(Array<dcomplex>& cv, Array<int>& ci) {
  sort<dcomplex>(cv);           // sort(Array<dcomplex>&)
  sort<int>(ci);                // sort(Array<int>&)
}
```

and

```
template<class U, class V> U convert(V v);
```

```
void g(double d) {
  int i = convert<int,double>(d);       // int convert(double)
  char c = convert<char,double>(d);     // char convert(double)
}
```

— *end example*]

1

A template argument list may be specified when referring to a specialization of a function template

— when a function is called,

— when the address of a function is taken, when a function initializes a reference to function, or when a pointer to member function is formed,

— in an explicit specialization,

— in an explicit instantiation, or

— in a friend declaration.

1

Trailing template arguments that can be deduced (14.2114.1) or obtained from default *template-argument*s may be omitted from the list of explicit *template-argument*s. A trailing template parameter pack (14.703.1) not otherwise deduced will be deduced to an empty sequence of template arguments. If all of the template arguments can be deduced, they may all be omitted; in this case, the empty template argument list `<>` itself may also be omitted. In contexts where deduction is done and fails, or in contexts where deduction is not done, if a template argument list is specified and it, along with any default template arguments, identifies a single function template specialization, then the *template-id* is an lvalue for the function template specialization. [*Example:*

```
template<class X, class Y> X f(Y);
template<class X, class Y, class ... Z> X g(Y);
void h() {
  int i = f<int>(5.6);          // Y is deduced to be double
  int j = f(5.6);               // ill-formed: X cannot be deduced
  f<void>(f<int, bool>);        // Y for outer f deduced to be
                                // int (*)(bool)
  f<void>(f<int>);              // ill-formed: f<int> does not denote a
                                // single function template specialization
  int k = g<int>(5.6);          // Y is deduced to be double, Z is deduced to an empty sequence
  f<void>(g<int, bool>);        // Y for outer f is deduced to be
                                // int (*)(bool), Z is deduced to an empty sequence
}
```

— *end example*]

1

[*Note:* An empty template argument list can be used to indicate that a given use refers to a specialization of a function template even when a non-template function (8.360.1) is visible that would otherwise be used. For example:

```
template <class T> int f(T);    // #1
int f(int);                     // #2
int k = f(1);                   // uses #2
int l = f<>(1);                 // uses #1
```

— *end note*]

1 Template arguments that are present shall be specified in the declaration order of their corresponding *template-parameter*s. The template argument list shall not specify more *template-argument*s than there are corresponding *template-parameter*s unless one of the *template-parameter*s is a template parameter pack. [*Example:*

```
template<class X, class Y, class Z> X f(Y,Z);
template<class ... Args> void f2();
void g() {
  f<int,const char*,double>("aa",3.0);
  f<int,const char*>("aa",3.0);        // Z is deduced to be double
  f<int>("aa",3.0);                // Y is deduced to be const char*, and
                                   // Z is deduced to be double
  f("aa",3.0);                     // error: X cannot be deduced
  f2<char, short, int, long>(); // OK
}
```

— *end example*]

2

Implicit conversions (Clause 4) will be performed on a function argument to convert it to the type of the corresponding function parameter if the parameter type contains no *template-parameter*s that participate in template argument deduction. [*Note:* Template parameters do not participate in template argument deduction if they are explicitly specified. For example,

```
template<class T> void f(T);

class Complex {
  Complex(double);
};

void g() {
  f<Complex>(1);                   // OK, means f<Complex>(Complex(1))
}
```

— *end note*]

1 [*Note:* Because the explicit template argument list follows the function template name, and because conversion member function templates and constructor member function templates are called without using a function name, there is no way to provide an explicit template argument list for these function templates. — *end note*]

2

[*Note:* For simple function names, argument dependent lookup (3.1092.1) applies even when the function name is not visible within the scope of the call. This is because the call still has the syntactic form of a function call (3.952.1). But when a function template with explicit template arguments is used, the call does not have the correct syntactic form unless there is a function template with that name visible at the point of the call. If no such name is visible, the call is not syntactically well-formed and argument-dependent lookup does not apply. If some such name is visible, argument dependent lookup applies and additional function templates may be found in other namespaces. [*Example:*

```
namespace A {
  struct B { };
  template<int X> void f(B);
}
namespace C {
```

```
  template<class T> void f(T t);
}
void g(A::B b) {
  f<3>(b);                      // ill-formed: not a function call
  A::f<3>(b);                   // well-formed
  C::f<3>(b);                   // ill-formed; argument dependent lookup
                                // applies only to unqualified names
  using C::f;
  f<3>(b);                      // well-formed because C::f is visible; then
                                // A::f is found by argument dependent lookup
}
```

— *end example*] — *end note*]

1

Template argument deduction can extend the sequence of template arguments corresponding to a template parameter pack, even when the sequence contains explicitly specified template arguments. [*Example:*

```
template<class ... Types> void f(Types ... values);

void g() {
  f<int*, float*>(0, 0, 0);     // Types is deduced to the sequence int*, float*, int
}
```

— *end example*]

14.2114.1 Template argument deduction [temp.deduct]

1

When a function template specialization is referenced, all of the template arguments shall have values. The values can be explicitly specified or, in some cases, be deduced from the use or obtained from default *template-arguments*. [*Example:*

```
void f(Array<dcomplex>& cv, Array<int>& ci) {
  sort(cv);                     // calls sort(Array<dcomplex>&)
  sort(ci);                     // calls sort(Array<int>&)
}
```

and

```
void g(double d) {
  int i = convert<int>(d);      // calls convert<int,double>(double)
  int c = convert<char>(d);     // calls convert<char,double>(double)
}
```

— *end example*]

1

When an explicit template argument list is specified, the template arguments must be compatible with the template parameter list and must result in a valid function type as described below; otherwise type deduction fails. Specifically, the following steps are performed when evaluating an explicitly specified template argument list with respect to a given function template:

— The specified template arguments must match the template parameters in kind (i.e., type, non-type, template). There must not be more arguments than there are parameters unless at least one parameter is a template parameter pack, and there shall be an argument for each non-pack parameter. Otherwise, type deduction fails.

— Non-type arguments must match the types of the corresponding non-type template parameters, or must be convertible to the types of the corresponding non-type parameters as specified in 14.364.1, otherwise type deduction fails.

— The specified template argument values are substituted for the corresponding template parameters as specified below.

1

After this substitution is performed, the function parameter type adjustments described in 8.360.1 are performed. [*Example:* A parameter type of "`void ()(const int, int[5])`" becomes "`void(*)(int,int*)`". — *end example*] [*Note:* A top-level qualifier in a function parameter declaration does not affect the function type but still affects the type of the function parameter variable within the function. — *end note*] [*Example:*

```
template <class T> void f(T t);
template <class X> void g(const X x);
template <class Z> void h(Z, Z*);

int main() {
  // #1: function type is f(int), t is non const
  f<int>(1);

  // #2: function type is f(int), t is const
  f<const int>(1);

  // #3: function type is g(int), x is const
  g<int>(1);

  // #4: function type is g(int), x is const
  g<const int>(1);

  // #5: function type is h(int, const int*)
  h<const int>(1,0);
}
```

— *end example*]

1

[*Note:* `f<int>(1)` and `f<const int>(1)` call distinct functions even though both of the functions called have the same function type. — *end note*]

1

The resulting substituted and adjusted function type is used as the type of the function template for template argument deduction. If a template argument has not been deduced and its corresponding template parameter has a default argument, the template argument is determined by substituting the template arguments determined for preceding template parameters into the default argument. If the substitution results in an invalid type, as described above, type deduction fails. [*Example:*

```
template <class T, class U = double>
void f(T t = 0, U u = 0);

void g() {
  f(1, 'c');        // f<int,char>(1,'c')
  f(1);             // f<int,double>(1,0)
  f();              // error: T cannot be deduced
  f<int>();         // f<int,double>(0,0)
  f<int,char>();    // f<int,char>(0,0)
```

```
}
```

— *end example*]

When all template arguments have been deduced or obtained from default template arguments, all uses of template parameters in the template parameter list of the template and the function type are replaced with the corresponding deduced or default argument values. If the substitution results in an invalid type, as described above, type deduction fails.

1

At certain points in the template argument deduction process it is necessary to take a function type that makes use of template parameters and replace those template parameters with the corresponding template arguments. This is done at the beginning of template argument deduction when any explicitly specified template arguments are substituted into the function type, and again at the end of template argument deduction when any template arguments that were deduced or obtained from default arguments are substituted.

1

The substitution occurs in all types and expressions that are used in the function type and in template parameter declarations. The expressions include not only constant expressions such as those that appear in array bounds or as nontype template arguments but also general expressions (i.e., non-constant expressions) inside `sizeof`, `decltype`, and other contexts that allow non-constant expressions. The substitution proceeds in lexical order and stops when a condition that causes deduction to fail is encountered. [*Note:* The equivalent substitution in exception specifications is done only when the *exception-specification* is instantiated, at which point a program is ill-formed if the substitution results in an invalid type or expression. — *end note*] [*Example:*

```
template <class T> struct A { using X = typename T::X; };
template <class T> typename T::X f(typename A<T>::X);
template <class T> void f(...) { }
template <class T> auto g(typename A<T>::X) -> typename T::X;
template <class T> void g(...) { }

void h() {
  f<int>(0); // OK, substituting return type causes deduction to fail
  g<int>(0); // error, substituting parameter type instantiates A<int>
}
```

— *end example*]

1

If a substitution results in an invalid type or expression, type deduction fails. An invalid type or expression is one that would be ill-formed, with a diagnostic required, if written using the substituted arguments. [*Note:* If no diagnostic is required, the program is still ill-formed. Access checking is done as part of the substitution process. — *end note*] Only invalid types and expressions in the immediate context of the function type and its template parameter types can result in a deduction failure. [*Note:* The evaluation of the substituted types and expressions can result in side effects such as the instantiation of class template specializations and/or function template specializations, the generation of implicitly-defined functions, etc. Such side effects are not in the "immediate context" and can result in the program being ill-formed. — *end note*]

[*Example:*

```
struct X { };
struct Y {
  Y(X){}
};
```

```
template <class T> auto f(T t1, T t2) -> decltype(t1 + t2); // #1
X f(Y, Y);  // #2

X x1, x2;
X x3 = f(x1, x2);  // deduction fails on #1 (cannot add X+X), calls #2
```

— *end example*]

[*Note:* Type deduction may fail for the following reasons:

— Attempting to instantiate a pack expansion containing multiple parameter packs of differing lengths.

— Attempting to create an array with an element type that is `void`, a function type, a reference type, or an abstract class type, or attempting to create an array with a size that is zero or negative. [*Example:*

```
template <class T> int f(T[5]);
int I = f<int>(0);
int j = f<void>(0);              // invalid array
```

— *end example*]

— Attempting to use a type that is not a class or enumeration type in a qualified name. [*Example:*

```
template <class T> int f(typename T::B*);
int i = f<int>(0);
```

— *end example*]

— Attempting to use a type in a *nested-name-specifier* of a *qualified-id* when that type does not contain the specified member, or

  — the specified member is not a type where a type is required, or

  — the specified member is not a template where a template is required, or

  — the specified member is not a non-type where a non-type is required.

[*Example:*

```
template <int I> struct X { };
template <template <class T> class> struct Z { };
template <class T> void f(typename T::Y*){}
template <class T> void g(X<T::N>*){}
template <class T> void h(Z<T::template TT>*){}
struct A {};
struct B { int Y; };
struct C {
  typedef int N;
};
struct D {
  typedef int TT;
};

int main() {
  // Deduction fails in each of these cases:
  f<A>(0);  // A does not contain a member Y
  f<B>(0);  // The Y member of B is not a type
  g<C>(0);  // The N member of C is not a non-type
  h<D>(0);  // The TT member of D is not a template
}
```

— *end example*]

— Attempting to create a pointer to reference type.

— Attempting to create a reference to `void`.

— Attempting to create "pointer to member of `T`" when `T` is not a class type. [*Example:*

```
template <class T> int f(int T::*);
int i = f<int>(0);
```

— *end example*]

— Attempting to give an invalid type to a non-type template parameter. [*Example:*

```
template <class T, T> struct S {};
template <class T> int f(S<T, T()>*);
struct X {};
int i0 = f<X>(0);
```

— *end example*]

— Attempting to perform an invalid conversion in either a template argument expression, or an expression used in the function declaration. [*Example:*

```
template <class T, T*> int f(int);
int i2 = f<int,1>(0);          // can't conv 1 to int*
```

— *end example*]

— Attempting to create a function type in which a parameter has a type of `void`, or in which the return type is a function type or array type.

— Attempting to create a function type in which a parameter type or the return type is an abstract class type (10.276).

— *end note*]

1

Except as described above, the use of an invalid value shall not cause type deduction to fail. [*Example:* In the following example 1000 is converted to `signed char` and results in an implementation-defined value as specified in (4.397). In other words, both templates are considered even though 1000, when converted to `signed char`, results in an implementation-defined value.

```
template <int> int f(int);
template <signed char> int f(int);
int i1 = f<1>(0);              // ambiguous
int i2 = f<1000>(0);           // ambiguous
```

— *end example*]

14.2261.0.1 Deducing template arguments from a function call [temp.deduct.call]

1

Template argument deduction is done by comparing each function template parameter type (call it `P`) with the type of the corresponding argument of the call (call it `A`) as described below. If removing references and cv-qualifiers from `P` gives `std::initializer_list<`P'`>` for some P' and the argument is an initializer list (8.2435.1), then deduction is performed instead for each element of the initializer list, taking P' as a function template parameter type and the initializer element as its argument. Otherwise, an initializer list argument causes the parameter to be considered a non-deduced context (14.2492.0.1). [*Example:*

```
template<class T> void f(std::initializer_list<T>);
f({1,2,3});                 // T deduced to int
f({1,"asdf"});              // error: T deduced to both int and const char*

template<class T> void g(T);
g({1,2,3});                 // error: no argument deduced for T
```

— *end example*] For a function parameter pack that occurs at the end of the *parameter-declaration-list*, the type `A` of each remaining argument of the call is compared with the type `P` of the *declarator-id* of the function parameter pack. Each comparison deduces template arguments for subsequent positions in the template parameter packs expanded by the function parameter pack. When a function parameter pack appears in a non-deduced context (14.2492.0.1), the type of that parameter pack is never deduced. [*Example:*

```
template<class ... Types> void f(Types& ...);
template<class T1, class ... Types> void g(T1, Types ...);
template<class T1, class ... Types> void g1(Types ..., T1);

void h(int x, float& y) {
  const int z = x;
  f(x, y, z);                  // Types is deduced to int, float, const int
  g(x, y, z);                  // T1 is deduced to int; Types is deduced to float, int
  g1(x, y, z);                 // error: Types is not deduced
  g1<int, int, int>(x, y, z);  // OK, no deduction occurs

}
```

— *end example*]

1

If `P` is not a reference type:

— If `A` is an array type, the pointer type produced by the array-to-pointer standard conversion (4.180) is used in place of `A` for type deduction; otherwise,

— If `A` is a function type, the pointer type produced by the function-to-pointer standard conversion (4.220) is used in place of `A` for type deduction; otherwise,

— If `A` is a cv-qualified type, the top level cv-qualifiers of `A`'s type are ignored for type deduction.

1

If `P` is a cv-qualified type, the top level cv-qualifiers of `P`'s type are ignored for type deduction. If `P` is a reference type, the type referred to by `P` is used for type deduction. If `P` is an rvalue reference to a cv-unqualified template parameter and the argument is an lvalue, the type "lvalue reference to `A`" is used in place of `A` for type deduction. [*Example:*

```
template <class T> int f(T&&);
template <class T> int g(const T&&);
int i;
int n1 = f(i);                  // calls f<int&>(int&)
int n2 = f(0);                  // calls f<int>(int&&)
int n3 = g(i);                  // error: would call g<int>(const int&&), which
                                // would bind an rvalue reference to an lvalue
```

— *end example*]

1

In general, the deduction process attempts to find template argument values that will make the deduced `A` identical to `A` (after the type `A` is transformed as described above). However, there are three cases that allow a difference:

— If the original `P` is a reference type, the deduced `A` (i.e., the type referred to by the reference) can be more cv-qualified than the transformed `A`.

— The transformed `A` can be another pointer or pointer to member type that can be converted to the deduced `A` via a qualification conversion (4.246).

— If `P` is a class and `P` has the form *simple-template-id*, then the transformed `A` can be a derived class of the deduced `A`. Likewise, if `P` is a pointer to a class of the form *simple-template-id*, the transformed `A` can be a pointer to a derived class pointed to by the deduced `A`.

[*Note:* as specified in 14.2075.1, implicit conversions will be performed on a function argument to convert it to the type of the corresponding function parameter if the parameter contains no *template-parameter*s that participate in template argument deduction. Such conversions are also allowed, in addition to the ones described in the preceding list. — *end note*]

1

These alternatives are considered only if type deduction would otherwise fail. If they yield more than one possible deduced `A`, the type deduction fails. [*Note:* If a *template-parameter* is not used in any of the function parameters of a function template, or is used only in a non-deduced context, its corresponding *template-argument* cannot be deduced from a function call and the *template-argument* must be explicitly specified. — *end note*]

1

When P is a function type, pointer to function type, or pointer to member function type:

— If the argument is an overload set containing one or more function templates, the parameter is treated as a non-deduced context.

— If the argument is an overload set (not containing function templates), trial argument deduction is attempted using each of the members of the set. If deduction succeeds for only one of the overload set members, that member is used as the argument value for the deduction. If deduction succeeds for more than one member of the overload set the parameter is treated as a non-deduced context.

1 [*Example:*

```
// Only one function of an overload set matches the call so the function
// parameter is a deduced context.
template <class T> int f(T (*p)(T));
int g(int);
int g(char);
int i = f(g);        // calls f(int (*)(int))
```

— *end example*]

2 [*Example:*

```
// Ambiguous deduction causes the second function parameter to be a
// non-deduced context.
template <class T> int f(T, T (*p)(T));
int g(int);
char g(char);
int i = f(1, g);     // calls f(int, int (*)(int))
```

— *end example*]

3 [*Example:*

```
// The overload set contains a template, causing the second function
// parameter to be a non-deduced context.
template <class T> int f(T, T (*p)(T));
char g(char);
template <class T> T g(T);
int i = f(1, g);     // calls f(int, int (*)(int))
```

— *end example*]

14.2407.0.1 Deducing template arguments taking the address of a function template [temp.deduct.funcaddr]

1

Template arguments can be deduced from the type specified when taking the address of an overloaded function (13.1084). The function template's function type and the specified type are used as the types of `P` and `A`, and the deduction is done as described in 14.2492.0.1.

1

A placeholder type (7.832.0.1) in the return type of a function template is a non-deduced context. If template argument deduction succeeds for such a function, the return type is determined from instantiation of the function body.

14.2413.0.1 Deducing conversion function template arguments [temp.deduct.conv]

1

Template argument deduction is done by comparing the return type of the conversion function template (call it `P`) with the type that is required as the result of the conversion (call it `A`; see 8.821, 13.396.0.1, and 13.413.0.1 for the determination of that type) as described in 14.2492.0.1.

1

If `P` is a reference type, the type referred to by `P` is used in place of `P` for type deduction and for any further references to or transformations of `P` in the remainder of this section.

1 If `A` is not a reference type:

(1.1) — If `P` is an array type, the pointer type produced by the array-to-pointer standard conversion (4.180) is used in place of `P` for type deduction; otherwise,

(1.2) — If `P` is a function type, the pointer type produced by the function-to-pointer standard conversion (4.220) is used in place of `P` for type deduction; otherwise,

(1.3) — If `P` is a cv-qualified type, the top level cv-qualifiers of `P`'s type are ignored for type deduction.

2 If `A` is a cv-qualified type, the top level cv-qualifiers of `A`'s type are ignored for type deduction. If `A` is a reference type, the type referred to by `A` is used for type deduction.

3

In general, the deduction process attempts to find template argument values that will make the deduced `A` identical to `A`. However, there are two cases that allow a difference:

— If the original `A` is a reference type, `A` can be more cv-qualified than the deduced `A` (i.e., the type referred to by the reference)

— The deduced `A` can be another pointer or pointer to member type that can be converted to `A` via a qualification conversion.

1 These alternatives are considered only if type deduction would otherwise fail. If they yield more than one possible deduced `A`, the type deduction fails.

2 When the deduction process requires a qualification conversion for a pointer or pointer to member type as described above, the following process is used to determine the deduced template argument values:

If `A` is a type

$$cv_{1,0} \text{ “pointer to ...” } cv_{1,n-1} \text{ “pointer to” } cv_{1,n}\, T1$$

and `P` is a type

$$cv_{2,0} \text{ “pointer to ...” } cv_{2,n-1} \text{ “pointer to” } cv_{2,n}\, T2$$

The cv-unqualified `T1` and `T2` are used as the types of `A` and `P` respectively for type deduction. [*Example:*

```
struct A {
  template <class T> operator T***();
};
A a;
const int * const * const * p1 = a;      // T is deduced as int, not const int
```

— *end example*]

14.2418.0.1 Deducing template arguments during partial ordering [temp.deduct.partial]

1 Template argument deduction is done by comparing certain types associated with the two function templates being compared.

2

Two sets of types are used to determine the partial ordering. For each of the templates involved there is the original function type and the transformed function type. [*Note:* The creation of the transformed type is described in 14.992.0.1. — *end note*] The deduction process uses the transformed type as the argument template and the original type of the other template as the parameter template. This process is done twice for each type involved in the partial ordering comparison: once using the transformed template-1 as the argument template and template-2 as the parameter template and again using the transformed template-2 as the argument template and template-1 as the parameter template.

1

The types used to determine the ordering depend on the context in which the partial ordering is done:

— In the context of a function call, the types used are those function parameter types for which the function call has arguments.[143]

— In the context of a call to a conversion function, the return types of the conversion function templates are used.

— In other contexts (14.992.0.1) the function template's function type is used.

1

Each type nominated above from the parameter template and the corresponding type from the argument template are used as the types of `P` and `A`.

1

143) Default arguments are not considered to be arguments in this context; they only become arguments after a function has been selected.

Before the partial ordering is done, certain transformations are performed on the types used for partial ordering:

— If `P` is a reference type, `P` is replaced by the type referred to.

— If `A` is a reference type, `A` is replaced by the type referred to.

1

If both `P` and `A` were reference types (before being replaced with the type referred to above), determine which of the two types (if any) is more cv-qualified than the other; otherwise the types are considered to be equally cv-qualified for partial ordering purposes. The result of this determination will be used below.

1 Remove any top-level cv-qualifiers:

(1.1) — If `P` is a cv-qualified type, `P` is replaced by the cv-unqualified version of `P`.

(1.2) — If `A` is a cv-qualified type, `A` is replaced by the cv-unqualified version of `A`.

2

If `A` was transformed from a function parameter pack and `P` is not a parameter pack, type deduction fails. Otherwise, using the resulting types `P` and `A`, the deduction is then done as described in 14.2492.0.1. If `P` is a function parameter pack, the type `A` of each remaining parameter type of the argument template is compared with the type `P` of the *declarator-id* of the function parameter pack. Each comparison deduces template arguments for subsequent positions in the template parameter packs expanded by the function parameter pack. If deduction succeeds for a given type, the type from the argument template is considered to be at least as specialized as the type from the parameter template. [*Example:*

```
template<class... Args>           void f(Args... args);           // #1
template<class T1, class... Args> void f(T1 a1, Args... args);    // #2
template<class T1, class T2>      void f(T1 a1, T2 a2);           // #3

f();                  // calls #1
f(1, 2, 3);           // calls #2
f(1, 2);              // calls #3; non-variadic template #3 is more
                      // specialized than the variadic templates #1 and #2
```

— *end example*]

1

If, for a given type, deduction succeeds in both directions (i.e., the types are identical after the transformations above) and both `P` and `A` were reference types (before being replaced with the type referred to above):

— if the type from the argument template was an lvalue reference and the type from the parameter template was not, the argument type is considered to be more specialized than the other; otherwise,

— if the type from the argument template is more cv-qualified than the type from the parameter template (as described above), the argument type is considered to be more specialized than the other; otherwise,

— neither type is more specialized than the other.

1

If for each type being considered a given template is at least as specialized for all types and more specialized for some set of types and the other template is not more specialized for any types or is not at least as specialized for any types, then the given template is more specialized than the other template. Otherwise, neither template is more specialized than the other.

1

In most cases, all template parameters must have values in order for deduction to succeed, but for partial ordering purposes a template parameter may remain without a value provided it is not used in the types being used for partial ordering. [*Note:* A template parameter used in a non-deduced context is considered used. — *end note*] [*Example:*

```
template <class T> T f(int);        // #1
template <class T, class U> T f(U); // #2
void g() {
  f<int>(1);        // calls #1
}
```

— *end example*]

1

[*Note:* Partial ordering of function templates containing template parameter packs is independent of the number of deduced arguments for those template parameter packs. — *end note*] [*Example:*

```
template<class ...> struct Tuple { };
template<class ... Types> void g(Tuple<Types ...>);                  // #1
template<class T1, class ... Types> void g(Tuple<T1, Types ...>);    // #2
template<class T1, class ... Types> void g(Tuple<T1, Types& ...>);   // #3

g(Tuple<>());                   // calls #1
g(Tuple<int, float>());         // calls #2
g(Tuple<int, float&>());        // calls #3
g(Tuple<int>());                // calls #3
```

— *end example*]

14.2492.0.1 Deducing template arguments from a type [temp.deduct.type]

1

Template arguments can be deduced in several different contexts, but in each case a type that is specified in terms of template parameters (call it `P`) is compared with an actual type (call it `A`), and an attempt is made to find template argument values (a type for a type parameter, a value for a non-type parameter, or a template for a template parameter) that will make `P`, after substitution of the deduced values (call it the deduced `A`), compatible with `A`.

1

In some cases, the deduction is done using a single set of types `P` and `A`, in other cases, there will be a set of corresponding types `P` and `A`. Type deduction is done independently for each `P/A` pair, and the deduced template argument values are then combined. If type deduction cannot be done for any `P/A` pair, or if for any pair the deduction leads to more than one possible set of deduced values, or if different pairs yield different deduced values, or if any template argument remains neither deduced nor explicitly specified, template argument deduction fails.

1 A given type `P` can be composed from a number of other types, templates, and non-type values:

(1.1) — A function type includes the types of each of the function parameters and the return type.

(1.2) — A pointer to member type includes the type of the class object pointed to and the type of the member pointed to.

(1.3) — A type that is a specialization of a class template (e.g., `A<int>`) includes the types, templates, and non-type values referenced by the template argument list of the specialization.

(1.4) — An array type includes the array element type and the value of the array bound.

2

In most cases, the types, templates, and non-type values that are used to compose `P` participate in template argument deduction. That is, they may be used to determine the value of a template argument, and the value so determined must be consistent with the values determined elsewhere. In certain contexts, however, the value does not participate in type deduction, but instead uses the values of template arguments that were either deduced elsewhere or explicitly specified. If a template parameter is used only in non-deduced contexts and is not explicitly specified, template argument deduction fails.

1

The non-deduced contexts are:

— The *nested-name-specifier* of a type that was specified using a *qualified-id*.

— The *expression* of a *decltype-specifier*.

— A non-type template argument or an array bound in which a subexpression references a template parameter.

— A template parameter used in the parameter type of a function parameter that has a default argument that is being used in the call for which argument deduction is being done.

— A function parameter for which argument deduction cannot be done because the associated function argument is a function, or a set of overloaded functions (13.1084), and one or more of the following apply:

  — more than one function matches the function parameter type (resulting in an ambiguous deduction), or

  — no function matches the function parameter type, or

  — the set of functions supplied as an argument contains one or more function templates.

— A function parameter for which the associated argument is an initializer list (8.2435.1) but the parameter does not have `std::initializer_list` or reference to possibly cv-qualified `std::initializer_list` type. [*Example:*

```
template<class T> void g(T);
g({1,2,3});                 // error: no argument deduced for T
```

— *end example*]

— A function parameter pack that does not occur at the end of the *parameter-declaration-list*.

1

When a type name is specified in a way that includes a non-deduced context, all of the types that comprise that type name are also non-deduced. However, a compound type can include both deduced and non-deduced types. [*Example:* If a type is specified as `A<T>::B<T2>`, both `T` and `T2` are non-deduced. Likewise, if a type is specified as `A<I+J>::X<T>`, `I`, `J`, and `T` are non-deduced. If a type is specified as `void f(typename A<T>::B, A<T>)`, the `T` in `A<T>::B` is non-deduced but the `T` in `A<T>` is deduced. — *end example*]

1 [*Example:* Here is an example in which different parameter/argument pairs produce inconsistent template argument deductions:

```
template<class T> void f(T x, T y) { /* ... */ }
struct A { /* ... */ };
struct B : A { /* ... */ };
void g(A a, B b) {
  f(a,b);           // error: T could be A or B
  f(b,a);           // error: T could be A or B
  f(a,a);           // OK: T is A
  f(b,b);           // OK: T is B
}
```

Here is an example where two template arguments are deduced from a single function parameter/argument pair. This can lead to conflicts that cause type deduction to fail:

```
template <class T, class U> void f(  T (*)( T, U, U )  );

int g1( int, float, float);
char g2( int, float, float);
int g3( int, char, float);

void r() {
  f(g1);            // OK: T is int and U is float
  f(g2);            // error: T could be char or int
  f(g3);            // error: U could be char or float
}
```

Here is an example where a qualification conversion applies between the argument type on the function call and the deduced template argument type:

```
template<class T> void f(const T*) { }
int* p;
void s() {
  f(p);             // f(const int*)
}
```

Here is an example where the template argument is used to instantiate a derived class type of the corresponding function parameter type:

```
template <class T> struct B { };
template <class T> struct D : public B<T> {};
struct D2 : public B<int> {};
template <class T> void f(B<T>&){}
void t() {
  D<int> d;
  D2     d2;
  f(d);             // calls f(B<int>&)
  f(d2);            // calls f(B<int>&)
}
```

— *end example*]

2

A template type argument `T`, a template template argument `TT` or a template non-type argument `i` can be deduced if `P` and `A` have one of the following forms:

```
T
cv-list T
T*
```

```
T&
T&&
T[integer-constant]
template-name<T>  (where template-name refers to a class template)
type(T)
T()
T(T)
T type::*
type T::*
T T::*
T (type::*)()
type (T::*)()
type (type::*)(T)
type (T::*)(T)
T (type::*)(T)
T (T::*)()
T (T::*)(T)
type[i]
template-name<i>  (where template-name refers to a class template)
TT<T>
TT<i>
TT<>
```

where `(T)` represents a *parameter-type-list* where at least one parameter type contains a `T`, and `()` represents a *parameter-type-list* where no parameter type contains a `T`. Similarly, `<T>` represents template argument lists where at least one argument contains a `T`, `<i>` represents template argument lists where at least one argument contains an `i` and `<>` represents template argument lists where no argument contains a `T` or an `i`.

1

If `P` has a form that contains `<T>` or `<i>`, then each argument $\mathtt{P}_i$ of the respective template argument list `P` is compared with the corresponding argument $\mathtt{A}_i$ of the corresponding template argument list of `A`. If the template argument list of `P` contains a pack expansion that is not the last template argument, the entire template argument list is a non-deduced context. If $\mathtt{P}_i$ is a pack expansion, then the pattern of $\mathtt{P}_i$ is compared with each remaining argument in the template argument list of `A`. Each comparison deduces template arguments for subsequent positions in the template parameter packs expanded by $\mathtt{P}_i$. During partial ordering (14.2418.0.1), if $\mathtt{A}_i$ was originally a pack expansion:

— if `P` does not contain a template argument corresponding to $\mathtt{A}_i$ then $\mathtt{A}_i$ is ignored;

— otherwise, if $\mathtt{P}_i$ is not a pack expansion, template argument deduction fails.

[*Example:*

```
template<class T1, class... Z> class S;                              // #1
template<class T1, class... Z> class S<T1, const Z&...> { };         // #2
template<class T1, class T2>   class S<T1, const T2&> { };           // #3
S<int, const int&> s;         // both #2 and #3 match; #3 is more specialized

template<class T, class... U>            struct A { };               // #1
template<class T1, class T2, class... U> struct A<T1, T2*, U...> { }; // #2
template<class T1, class T2>             struct A<T1, T2> { };       // #3
template struct A<int, int*>; // selects #2
```

— *end example*]

1

Similarly, if P has a form that contains (T), then each parameter type P_i of the respective *parameter-type-list* of P is compared with the corresponding parameter type A_i of the corresponding *parameter-type-list* of A. If P and A are function types that originated from deduction when taking the address of a function template (14.2407.0.1) or when deducing template arguments from a function declaration (14.2655.0.1) and P_i and A_i are parameters of the top-level *parameter-type-list* of P and A, respectively, P_i is adjusted if it is an rvalue reference to a cv-unqualified template parameter and A_i is an lvalue reference, in which case the type of P_i is changed to be the template parameter type (i.e., T&& is changed to simply T). [*Note:* As a result, when P_i is T&& and A_i is X&, the adjusted P_i will be T, causing T to be deduced as X&. — *end note*]
[*Example:*

```
template <class T> void f(T&&);
template <> void f(int&) { }  // #1
template <> void f(int&&) { } // #2
void g(int i) {
  f(i);                       // calls f<int&>(int&), i.e., #1
  f(0);                       // calls f<int>(int&&), i.e., #2
}
```

— *end example*]

If the *parameter-declaration* corresponding to P_i is a function parameter pack, then the type of its *declarator-id* is compared with each remaining parameter type in the *parameter-type-list* of A. Each comparison deduces template arguments for subsequent positions in the template parameter packs expanded by the function parameter pack. During partial ordering (14.2418.0.1), if A_i was originally a function parameter pack:

— if P does not contain a function parameter type corresponding to A_i then A_i is ignored;

— otherwise, if P_i is not a function parameter pack, template argument deduction fails.

[*Example:*

```
template<class T, class... U> void f(T*, U...) { }    // #1
template<class T>             void f(T) { }           // #2
template void f(int*);      // selects #1
```

— *end example*]

1 These forms can be used in the same way as T is for further composition of types. [*Example:*

```
X<int> (*)(char[6])
```

is of the form

```
template-name<T> (*)(type[i])
```

which is a variant of

```
type (*)(T)
```

where type is X<int> and T is char[6]. — *end example*]

2 Template arguments cannot be deduced from function arguments involving constructs other than the ones specified above.

3

A template type argument cannot be deduced from the type of a non-type *template-argument*.

1 [*Example:*

```
template<class T, T i> void f(double a[10][i]);
int v[10][20];
f(v);                // error: argument for template-parameter T cannot be deduced
```

— *end example*]

2 [*Note:* Except for reference and pointer types, a major array bound is not part of a function parameter type and cannot be deduced from an argument:

```
template<int i> void f1(int a[10][i]);
template<int i> void f2(int a[i][20]);
template<int i> void f3(int (&a)[i][20]);

void g() {
  int v[10][20];
  f1(v);             // OK: i deduced to be 20
  f1<20>(v);         // OK
  f2(v);             // error: cannot deduce template-argument i
  f2<10>(v);         // OK
  f3(v);             // OK: i deduced to be 10
}
```

3

If, in the declaration of a function template with a non-type template parameter, the non-type template parameter is used in a subexpression in the function parameter list, the expression is a non-deduced context as specified above. [*Example:*

```
template <int i> class A { /* ... */ };
template <int i> void g(A<i+1>);
template <int i> void f(A<i>, A<i+1>);
void k() {
  A<1> a1;
  A<2> a2;
  g(a1);             // error: deduction fails for expression i+1
  g<0>(a1);          // OK
  f(a1, a2);         // OK
}
```

— *end example*] — *end note*] [*Note:* Template parameters do not participate in template argument deduction if they are used only in non-deduced contexts. For example,

```
template<int i, typename T>
T deduce(typename A<T>::X x,     // T is not deduced here
  T t,                           // but T is deduced here
  typename B<i>::Y y);           // i is not deduced here
A<int> a;
B<77>  b;

int    x = deduce<77>(a.xm, 62, b.ym);
// T is deduced to be int, a.xm must be convertible to
// A<int>::X
// i is explicitly specified to be 77, b.ym must be convertible
// to B<77>::Y
```

— *end note*]

1

If `P` has a form that contains `<i>`, and if the type of the corresponding value of `A` differs from the type of `i`, deduction fails. If `P` has a form that contains `[i]`, and if the type of `i` is not an integral type, deduction fails.[144] [*Example:*

```
template<int i> class A { /* ... */ };
template<short s> void f(A<s>);
void k1() {
  A<1> a;
  f(a);             // error: deduction fails for conversion from int to short
  f<1>(a);          // OK
}

template<const short cs> class B { };
template<short s> void g(B<s>);
void k2() {
  B<1> b;
  g(b);             // OK: cv-qualifiers are ignored on template parameter types
}
```

— *end example*]

1 A *template-argument* can be deduced from a function, pointer to function, or pointer to member function type.

[*Example:*

```
template<class T> void f(void(*)(T,int));
template<class T> void foo(T,int);
void g(int,int);
void g(char,int);

void h(int,int,int);
void h(char,int);
int m() {
  f(&g);            // error: ambiguous
  f(&h);            // OK: void h(char,int) is a unique match
  f(&foo);          // error: type deduction fails because foo is a template
}
```

— *end example*]

2 A template *type-parameter* cannot be deduced from the type of a function default argument. [*Example:*

```
template <class T> void f(T = 5, T = 7);
void g() {
  f(1);             // OK: call f<int>(1,7)
  f();              // error: cannot deduce T
  f<int>();         // OK: call f<int>(5,7)
}
```

— *end example*]

1 The *template-argument* corresponding to a template *template-parameter* is deduced from the type of the *template-argument* of a class template specialization used in the argument list of a function call. [*Example:*

144) Although the *template-argument* corresponding to a *template-parameter* of type `bool` may be deduced from an array bound, the resulting value will always be `true` because the array bound will be non-zero.

```
template <template <class T> class X> struct A { };
template <template <class T> class X> void f(A<X>) { }
template<class T> struct B { };
A<B> ab;
f(ab);              // calls f(A<B>)
```

— *end example*]

2 [*Note:* Template argument deduction involving parameter packs (14.703.1) can deduce zero or more arguments for each parameter pack. — *end note*] [*Example:*

```
template<class> struct X { };
template<class R, class ... ArgTypes> struct X<R(int, ArgTypes ...)> { };
template<class ... Types> struct Y { };
template<class T, class ... Types> struct Y<T, Types& ...> { };

template<class ... Types> int f(void (*)(Types ...));
void g(int, float);

X<int> x1;                      // uses primary template
X<int(int, float, double)> x2;  // uses partial specialization; ArgTypes contains float, double
X<int(float, int)> x3;          // uses primary template
Y<> y1;                         // use primary template; Types is empty
Y<int&, float&, double&> y2;    // uses partial specialization; T is int&, Types contains float, double
Y<int, float, double> y3;       // uses primary template; Types contains int, float, double
int fv = f(g);                  // OK; Types contains int, float
```

— *end example*]

14.2655.0.1 Deducing template arguments from a function declaration [temp.deduct.decl]

1

In a declaration whose *declarator-id* refers to a specialization of a function template, template argument deduction is performed to identify the specialization to which the declaration refers. Specifically, this is done for explicit instantiations (14.1841.1), explicit specializations (14.1921.1), and certain friend declarations (14.755.1). This is also done to determine whether a deallocation function template specialization matches a placement `operator new` (3.2428.0.1, 5.2221.1). In all these cases, `P` is the type of the function template being considered as a potential match and `A` is either the function type from the declaration or the type of the deallocation function that would match the placement `operator new` as described in 5.2221.1. The deduction is done as described in 14.2492.0.1.

1

If, for the set of function templates so considered, there is either no match or more than one match after partial ordering has been considered (14.992.0.1), deduction fails and, in the declaration cases, the program is ill-formed.

14.2662.1 Overload resolution [temp.over]

1

A function template can be overloaded either by (non-template) functions of its name or by (other) function templates of the same name. When a call to that name is written (explicitly, or implicitly using the operator notation), template argument deduction (14.2114.1) and checking of any explicit template arguments (14.326) are performed for each function template to find the template argument values (if any) that can be used with that function template to instantiate a function template specialization that can be invoked with the call arguments. For each function template, if the argument deduction and checking succeeds, the *template-arguments* (deduced and/or explicit) are used to synthesize the declaration of a single function template

specialization which is added to the candidate functions set to be used in overload resolution. If, for a given function template, argument deduction fails, no such function is added to the set of candidate functions for that template. The complete set of candidate functions includes all the synthesized declarations and all of the non-template overloaded functions of the same name. The synthesized declarations are treated like any other functions in the remainder of overload resolution, except as explicitly noted in 13.562.1.[145]

[*Example:*

```
template<class T> T max(T a, T b) { return a>b?a:b; }

void f(int a, int b, char c, char d) {
  int m1 = max(a,b);            // max(int a, int b)
  char m2 = max(c,d);           // max(char a, char b)
  int m3 = max(a,c);            // error: cannot generate max(int,char)
}
```

1 Adding the non-template function

```
int max(int,int);
```

to the example above would resolve the third call, by providing a function that could be called for `max(a,c)` after using the standard conversion of `char` to `int` for `c`.

2 Here is an example involving conversions on a function argument involved in *template-argument* deduction:

```
template<class T> struct B { /* ... */ };
template<class T> struct D : public B<T> { /* ... */ };
template<class T> void f(B<T>&);

void g(B<int>& bi, D<int>& di) {
  f(bi);            // f(bi)
  f(di);            // f((B<int>&)di)
}
```

3 Here is an example involving conversions on a function argument not involved in *template-parameter* deduction:

```
template<class T> void f(T*,int);       // #1
template<class T> void f(T,char);       // #2

void h(int* pi, int i, char c) {
  f(pi,i);          // #1: f<int>(pi,i)
  f(pi,c);          // #2: f<int*>(pi,c)

  f(i,c);           // #2: f<int>(i,c);
  f(i,i);           // #2: f<int>(i,char(i))
}
```

— *end example*]

4

Only the signature of a function template specialization is needed to enter the specialization in a set of candidate functions. Therefore only the function template declaration is needed to resolve a call for which a template specialization is a candidate. [*Example:*

145) The parameters of function template specializations contain no template parameter types. The set of conversions allowed on deduced arguments is limited, because the argument deduction process produces function templates with parameters that either match the call arguments exactly or differ only in ways that can be bridged by the allowed limited conversions. Non-deduced arguments allow the full range of conversions. Note also that 13.562.1 specifies that a non-template function will be given preference over a template specialization if the two functions are otherwise equally good candidates for an overload match.

```
template<class T> void f(T);     // declaration

void g() {
  f("Annemarie");                // call of f<const char*>
}
```

1 The call of `f` is well-formed even if the template `f` is only declared and not defined at the point of the call. The program will be ill-formed unless a specialization for `f<const char*>`, either implicitly or explicitly generated, is present in some translation unit. — *end example*]

15 Exception handling [except]

1

Exception handling provides a way of transferring control and information from a point in the execution of a thread to an exception handler associated with a point previously passed by the execution. A handler will be invoked only by throwing an exception in code executed in the handler's try block or in functions called from the handler's try block.

try-block:
`try` *compound-statement handler-seq*

function-try-block:
`try` *ctor-initializer*$_{opt}$ *compound-statement handler-seq*

handler-seq:
handler handler-seq$_{opt}$

handler:
`catch (` *exception-declaration* `)` *compound-statement*

exception-declaration:
attribute-specifier-seq$_{opt}$ *type-specifier-seq declarator*
attribute-specifier-seq$_{opt}$ *type-specifier-seq abstract-declarator*$_{opt}$
`...`

throw-expression:
`throw` *assignment-expression*$_{opt}$

The optional *attribute-specifier-seq* in an *exception-declaration* appertains to the parameter of the catch clause (15.146).

1 A *try-block* is a *statement* (Clause 6). A *throw-expression* is of type `void`. [*Note:* Within this Clause "try block" is taken to mean both *try-block* and *function-try-block*. — *end note*]

2

A `goto` or `switch` statement shall not be used to transfer control into a try block or into a handler. [*Example:*

```
void f() {
  goto l1;          // Ill-formed
  goto l2;          // Ill-formed
  try {
    goto l1;        // OK
    goto l2;        // Ill-formed
    l1: ;
  } catch (...) {
    l2: ;
    goto l1;        // Ill-formed
    goto l2;        // OK
  }
}
```

— *end example*] A `goto`, `break`, `return`, or `continue` statement can be used to transfer control out of a try block or handler. When this happens, each variable declared in the try block will be destroyed in the context that directly contains its declaration. [*Example:*

```
lab:  try {
```

```
T1 t1;
try {
  T2 t2;
  if (condition)
    goto lab;
  } catch(...) { /* handler 2 */ }
} catch(...) { /* handler 1 */ }
```

Here, executing `goto lab;` will destroy first `t2`, then `t1`, assuming the *condition* does not declare a variable. Any exception raised while destroying `t2` will result in executing *handler 2*; any exception raised while destroying `t1` will result in executing *handler 1*. — *end example*]

1

A *function-try-block* associates a *handler-seq* with the *ctor-initializer*, if present, and the *compound-statement*. An exception thrown during the execution of the *compound-statement* or, for constructors and destructors, during the initialization or destruction, respectively, of the class's subobjects, transfers control to a handler in a *function-try-block* in the same way as an exception thrown during the execution of a *try-block* transfers control to other handlers. [*Example:*

```
int f(int);
class C {
  int i;
  double d;
public:
  C(int, double);
};

C::C(int ii, double id)
try : i(f(ii)), d(id) {
    // constructor statements
}
catch (...) {
    // handles exceptions thrown from the ctor-initializer
    // and from the constructor statements
}
```

— *end example*]

15.7 Throwing an exception [except.throw]

1

Throwing an exception transfers control to a handler. [*Note:* An exception can be thrown from one of the following contexts: *throw-expression* (see below), allocation functions (3.2361.0.1), `dynamic_cast` (5.1295.1), `typeid` (5.1393.1), *new-expression* (5.2221.1), and standard library functions (17.14.0.1). — *end note*] An object is passed and the type of that object determines which handlers can catch it. [*Example:*

```
throw "Help!";
```

can be caught by a *handler* of `const char*` type:

```
try {
    // ...
}
catch(const char* p) {
    // handle character string exceptions here
}
```

and

```
class Overflow {
public:
    Overflow(char,double,double);
};

void f(double x) {
    throw Overflow('+',x,3.45e107);
}
```

can be caught by a handler for exceptions of type `Overflow`

```
try {
    f(1.2);
} catch(Overflow& oo) {
    // handle exceptions of type Overflow here
}
```

— *end example*]

1

When an exception is thrown, control is transferred to the nearest handler with a matching type (15.146); "nearest" means the handler for which the *compound-statement* or *ctor-initializer* following the `try` keyword was most recently entered by the thread of control and not yet exited.

1

Throwing an exception copy-initializes (8.821, 12.1577) a temporary object, called the *exception object*. The temporary is an lvalue and is used to initialize the variable declared in the matching *handler* (15.146). If the type of the exception object would be an incomplete type or a pointer to an incomplete type other than (possibly cv-qualified) `void` the program is ill-formed. Evaluating a *throw-expression* with an operand throws an exception; the type of the exception object is determined by removing any top-level *cv-qualifiers* from the static type of the operand and adjusting the type from "array of `T`" or "function returning `T`" to "pointer to `T`" or "pointer to function returning `T`," respectively.

1

The memory for the exception object is allocated in an unspecified way, except as noted in 3.2361.0.1. If a handler exits by rethrowing, control is passed to another handler for the same exception. The exception object is destroyed after either the last remaining active handler for the exception exits by any means other than rethrowing, or the last object of type `std::exception_ptr` (18.254.1) that refers to the exception object is destroyed, whichever is later. In the former case, the destruction occurs when the handler exits, immediately after the destruction of the object declared in the *exception-declaration* in the handler, if any. In the latter case, the destruction occurs before the destructor of `std::exception_ptr` returns. The implementation may then deallocate the memory for the exception object; any such deallocation is done in an unspecified way. [*Note:* a thrown exception does not propagate to other threads unless caught, stored, and rethrown using appropriate library functions; see 18.254.1 and 30.179. — *end note*]

1

When the thrown object is a class object, the constructor selected for the copy-initialization and the destructor shall be accessible, even if the copy/move operation is elided (12.1577).

1 An exception is considered caught when a handler for that exception becomes active (15.146). [*Note:* An exception can have active handlers and still be considered uncaught if it is rethrown. — *end note*]

2

If the exception handling mechanism, after completing the initialization of the exception object but before the activation of a handler for the exception, calls a function that exits via an exception, `std::terminate` is called (15.308.1). [*Example:*

```
struct C {
  C() { }
  C(const C&) {
    if (std::uncaught_exception()) {
      throw 0;        // throw during copy to handler's exception-declaration object (15.146)
    }
  }
};

int main() {
  try {
    throw C();        // calls std::terminate() if construction of the handler's
                      // exception-declaration object is not elided (12.1577)
  } catch(C) { }
}
```

— *end example*]

1

A *throw-expression* with no operand rethrows the currently handled exception (15.146). The exception is reactivated with the existing exception object; no new exception object is created. The exception is no longer considered to be caught; therefore, the value of `std::uncaught_exception()` will again be `true`. [*Example:* code that must be executed because of an exception yet cannot completely handle the exception can be written like this:

```
try {
    // ...
} catch (...) {       // catch all exceptions
  // respond (partially) to exception
  throw;              // pass the exception to some
                      // other handler
}
```

— *end example*]

1

If no exception is presently being handled, executing a *throw-expression* with no operand calls `std::terminate()` (15.308.1).

15.89 Constructors and destructors [except.ctor]

1

As control passes from the point where an exception is thrown to a handler, destructors are invoked for all automatic objects constructed since the try block was entered. The automatic objects are destroyed in the reverse order of the completion of their construction.

1

An object of any storage duration whose initialization or destruction is terminated by an exception will have destructors executed for all of its fully constructed subobjects (excluding the variant members of a union-like class), that is, for subobjects for which the principal constructor (12.1026.1) has completed execution and the destructor has not yet begun execution. Similarly, if the non-delegating constructor for an object

has completed execution and a delegating constructor for that object exits with an exception, the object's destructor will be invoked. If the object was allocated in a *new-expression*, the matching deallocation function (3.2428.0.1, 5.2221.1, 12.988), if any, is called to free the storage occupied by the object.

1

The process of calling destructors for automatic objects constructed on the path from a try block to the point where an exception is thrown is called "*stack unwinding*." If a destructor called during stack unwinding exits with an exception, `std::terminate` is called (15.308.1). [*Note:* So destructors should generally catch exceptions and not let them propagate out of the destructor. — *end note*]

15.146 Handling an exception [except.handle]

1

The *exception-declaration* in a *handler* describes the type(s) of exceptions that can cause that *handler* to be entered. The *exception-declaration* shall not denote an incomplete type, an abstract class type, or an rvalue reference type. The *exception-declaration* shall not denote a pointer or reference to an incomplete type, other than `void*`, `const void*`, `volatile void*`, or `const volatile void*`.

1 A handler of type "array of `T`" or "function returning `T`" is adjusted to be of type "pointer to `T`" or "pointer to function returning `T`", respectively.

2

A *handler* is a match for an exception object of type `E` if

— The *handler* is of type *cv* `T` or *cv* `T&` and `E` and `T` are the same type (ignoring the top-level *cv-qualifiers*), or

— the *handler* is of type *cv* `T` or *cv* `T&` and `T` is an unambiguous public base class of `E`, or

— the *handler* is of type *cv* `T` or `const T&` where `T` is a pointer type and `E` is a pointer type that can be converted to `T` by either or both of

  — a standard pointer conversion (4.579) not involving conversions to pointers to private or protected or ambiguous classes

  — a qualification conversion, or

— the *handler* is of type *cv* `T` or `const T&` where `T` is a pointer or pointer to member type and `E` is `std::nullptr_t`.

[*Note:* A *throw-expression* whose operand is an integer literal with value zero does not match a handler of pointer or pointer to member type. — *end note*]

[*Example:*

```
class Matherr { /* ... */ virtual void vf(); };
class Overflow: public Matherr { /* ... */ };
class Underflow: public Matherr { /* ... */ };
class Zerodivide: public Matherr { /* ... */ };

void f() {
  try {
    g();
  } catch (Overflow oo) {
        // ...
  } catch (Matherr mm) {
        // ...
```

```
  }
}
```

Here, the `Overflow` handler will catch exceptions of type `Overflow` and the `Matherr` handler will catch exceptions of type `Matherr` and of all types publicly derived from `Matherr` including exceptions of type `Underflow` and `Zerodivide`. — *end example*]

1

The handlers for a try block are tried in order of appearance. That makes it possible to write handlers that can never be executed, for example by placing a handler for a derived class after a handler for a corresponding base class.

1

A `...` in a handler's *exception-declaration* functions similarly to `...` in a function parameter declaration; it specifies a match for any exception. If present, a `...` handler shall be the last handler for its try block.

1

If no match is found among the handlers for a try block, the search for a matching handler continues in a dynamically surrounding try block of the same thread.

1

A handler is considered active when initialization is complete for the parameter (if any) of the catch clause. [*Note:* The stack will have been unwound at that point. — *end note*] Also, an implicit handler is considered active when `std::terminate()` or `std::unexpected()` is entered due to a throw. A handler is no longer considered active when the catch clause exits or when `std::unexpected()` exits after being entered due to a throw.

1

The exception with the most recently activated handler that is still active is called the *currently handled exception*.

1

If no matching handler is found, the function `std::terminate()` is called; whether or not the stack is unwound before this call to `std::terminate()` is implementation-defined (15.308.1).

1

Referring to any non-static member or base class of an object in the handler for a *function-try-block* of a constructor or destructor for that object results in undefined behavior.

1

The fully constructed base classes and members of an object shall be destroyed before entering the handler of a *function-try-block* of a constructor for that object. Similarly, if a delegating constructor for an object exits with an exception after the non-delegating constructor for that object has completed execution, the object's destructor shall be executed before entering the handler of a *function-try-block* of a constructor for that object. The base classes and non-variant members of an object shall be destroyed before entering the handler of a *function-try-block* of a destructor for that object (12.807).

1 The scope and lifetime of the parameters of a function or constructor extend into the handlers of a *function-try-block*.

2

Exceptions thrown in destructors of objects with static storage duration or in constructors of namespace-scope objects with static storage duration are not caught by a *function-try-block* on `main()`. Exceptions thrown in destructors of objects with thread storage duration or in constructors of namespace-scope objects

with thread storage duration are not caught by a *function-try-block* on the initial function of the thread.

1

If a return statement appears in a handler of the *function-try-block* of a constructor, the program is ill-formed.

1

The currently handled exception is rethrown if control reaches the end of a handler of the *function-try-block* of a constructor or destructor. Otherwise, a function returns when control reaches the end of a handler for the *function-try-block* (6.270.2). Flowing off the end of a *function-try-block* is equivalent to a `return` with no value; this results in undefined behavior in a value-returning function (6.270.2).

1

The variable declared by the *exception-declaration*, of type *cv* `T` or *cv* `T&`, is initialized from the exception object, of type `E`, as follows:

— if `T` is a base class of `E`, the variable is copy-initialized (8.821) from the corresponding base class subobject of the exception object;

— otherwise, the variable is copy-initialized (8.821) from the exception object.

The lifetime of the variable ends when the handler exits, after the destruction of any automatic objects initialized within the handler.

1

When the handler declares an object, any changes to that object will not affect the exception object. When the handler declares a reference to an object, any changes to the referenced object are changes to the exception object and will have effect should that object be rethrown.

15.228 Exception specifications [except.spec]

1

A function declaration lists exceptions that its function might directly or indirectly throw by using an *exception-specification* as a suffix of its declarator.

exception-specification:
  dynamic-exception-specification
  noexcept-specification

dynamic-exception-specification:
  `throw (` *type-id-list*$_{opt}$`)`

type-id-list:
  type-id `...`$_{opt}$
  type-id-list `,` *type-id* `...`$_{opt}$

noexcept-specification:
  `noexcept (` *constant-expression* `)`
  `noexcept`

In a *noexcept-specification*, the *constant-expression*, if supplied, shall be a constant expression (5.3643) that is contextually converted to `bool` (Clause 4). A *noexcept-specification* `noexcept` is equivalent to `noexcept(true)`. A `(` token that follows `noexcept` is part of the *noexcept-specification* and does not commence an initializer (8.821).

1

An *exception-specification* shall appear only on a function declarator for a function type, pointer to function type, reference to function type, or pointer to member function type that is the top-level type of a declaration or definition, or on such a type appearing as a parameter or return type in a function declarator. An *exception-specification* shall not appear in a typedef declaration or *alias-declaration*. [*Example:*

```
void f() throw(int);                    // OK
void (*fp)() throw (int);               // OK
void g(void pfa() throw(int));          // OK
typedef int (*pf)() throw(int);         // ill-formed
```

— *end example*]

A type denoted in an *exception-specification* shall not denote an incomplete type or an rvalue reference type. A type denoted in an *exception-specification* shall not denote a pointer or reference to an incomplete type, other than *cv* `void*`. A type *cv* `T`, "array of `T`", or "function returning `T`" denoted in an *exception-specification* is adjusted to type `T`, "pointer to `T`", or "pointer to function returning `T`", respectively.

1

Two *exception-specification*s are *compatible* if:

— both are non-throwing (see below), regardless of their form,

— both have the form `noexcept(`*constant-expression*`)` and the *constant-expression*s are equivalent, or

— both are *dynamic-exception-specification*s that have the same set of adjusted types.

1

If any declaration of a function has an *exception-specification* that is not a *noexcept-specification* allowing all exceptions, all declarations, including the definition and any explicit specialization, of that function shall have a compatible *exception-specification*. If any declaration of a pointer to function, reference to function, or pointer to member function has an *exception-specification*, all occurrences of that declaration shall have a compatible *exception-specification* In an explicit instantiation an *exception-specification* may be specified, but is not required. If an *exception-specification* is specified in an explicit instantiation directive, it shall be compatible with the *exception-specification*s of other declarations of that function. A diagnostic is required only if the *exception-specification*s are not compatible within a single translation unit.

1

If a virtual function has an *exception-specification*, all declarations, including the definition, of any function that overrides that virtual function in any derived class shall only allow exceptions that are allowed by the *exception-specification* of the base class virtual function. [*Example:*

```
struct B {
  virtual void f() throw (int, double);
  virtual void g();
};

struct D: B {
  void f();                     // ill-formed
  void g() throw (int);         // OK
};
```

The declaration of `D::f` is ill-formed because it allows all exceptions, whereas `B::f` allows only `int` and `double`. — *end example*] A similar restriction applies to assignment to and initialization of pointers to functions, pointers to member functions, and references to functions: the target entity shall allow at least the exceptions allowed by the source value in the assignment or initialization. [*Example:*

```
class A { /* ... */ };
void (*pf1)();       // no exception specification
void (*pf2)() throw(A);

void f() {
```

```
  pf1 = pf2;          // OK: pf1 is less restrictive
  pf2 = pf1;          // error: pf2 is more restrictive
}
```

— *end example*]

1 In such an assignment or initialization, *exception-specification*s on return types and parameter types shall be compatible. In other assignments or initializations, *exception-specification*s shall be compatible.

2

An *exception-specification* can include the same type more than once and can include classes that are related by inheritance, even though doing so is redundant. [*Note:* An *exception-specification* can also include the class `std::bad_exception` (18.247.1). — *end note*]

1

A function is said to *allow* an exception of type `E` if the *constant-expression* in its *noexcept-specification* evaluates to `false` or its *dynamic-exception-specification* contains a type `T` for which a handler of type `T` would be a match (15.146) for an exception of type `E`.

1

Whenever an exception is thrown and the search for a handler (15.146) encounters the outermost block of a function with an *exception-specification* that does not allow the exception, then,

— if the *exception-specification* is a *dynamic-exception-specification*, the function `std::unexpected()` is called (15.370.1),

— otherwise, the function `std::terminate()` is called (15.308.1).

[*Example:*

```
class X { };
class Y { };
class Z: public X { };
class W { };

void f() throw (X, Y) {
  int n = 0;
  if (n) throw X();          // OK
  if (n) throw Z();          // also OK
  throw W();                 // will call std::unexpected()
}
```

— *end example*]

[*Note:* A function can have multiple declarations with different non-throwing *exception-specification*s; for this purpose, the one on the function definition is used. — *end note*]

1 The function `unexpected()` may throw an exception that will satisfy the *exception-specification* for which it was invoked, and in this case the search for another handler will continue at the call of the function with this *exception-specification* (see 15.370.1), or it may call `std::terminate()`.

2

An implementation shall not reject an expression merely because when executed it throws or might throw an exception that the containing function does not allow. [*Example:*

```
extern void f() throw(X, Y);

void g() throw(X) {
```

```
  f();                          // OK
}
```

the call to `f` is well-formed even though when called, `f` might throw exception `Y` that `g` does not allow. — *end example*]

1

A function with no *exception-specification* or with an *exception-specification* of the form `noexcept(`*constant-expression* `)` where the *constant-expression* yields `false` allows all exceptions. An *exception-specification* is *non-throwing* if it is of the form `throw()`, `noexcept`, or `noexcept(`*constant-expression* `)` where the *constant-expression* yields `true`. A function with a non-throwing *exception-specification* does not allow any exceptions.

1 An *exception-specification* is not considered part of a function's type.

2

An inheriting constructor (12.2400) and an implicitly declared special member function (Clause 12) have an *exception-specification*. If `f` is an inheriting constructor or an implicitly declared default constructor, copy constructor, move constructor, destructor, copy assignment operator, or move assignment operator, its implicit *exception-specification* specifies the *type-id* `T` if and only if `T` is allowed by the *exception-specification* of a function directly invoked by `f`'s implicit definition; `f` allows all exceptions if any function it directly invokes allows all exceptions, and `f` has the *exception-specification* `noexcept(true)` if every function it directly invokes allows no exceptions. [*Note:* It follows that `f` has the *exception-specification* `noexcept(true)` if it invokes no other functions. — *end note*] [*Note:* An instantiation of an inheriting constructor template has an implied *exception-specification* as if it were a non-template inheriting constructor. — *end note*] [*Example:*

```
struct A {
  A();
  A(const A&) throw();
  A(A&&) throw();
  ~A() throw(X);
};
struct B {
  B() throw();
  B(const B&) = default; // Declaration of B::B(const B&) noexcept(true)
  B(B&&) throw(Y);
  ~B() throw(Y);
};
struct D : public A, public B {
    // Implicit declaration of D::D();
    // Implicit declaration of D::D(const D&) noexcept(true);
    // Implicit declaration of D::D(D&&) throw(Y);
    // Implicit declaration of D::~D() throw(X, Y);
};
```

Furthermore, if `A::~A()` or `B::~B()` were virtual, `D::~D()` would not be as restrictive as that of `A::~A`, and the program would be ill-formed since a function that overrides a virtual function from a base class shall have an *exception-specification* at least as restrictive as that in the base class. — *end example*]

1 A deallocation function (3.2428.0.1) with no explicit *exception-specification* is treated as if it were specified with `noexcept(true)`.

2 An *exception-specification* is considered to be *needed* when:

(2.1) — in an expression, the function is the unique lookup result or the selected member of a set of overloaded functions (3.923, 13.88, 13.1084);

(2.2) — the function is odr-used (3.91) or, if it appears in an unevaluated operand, would be odr-used if the expression were potentially-evaluated;

(2.3) — the *exception-specification* is compared to that of another declaration (e.g., an explicit specialization or an overriding virtual function);

(2.4) — the function is defined; or

(2.5) — the *exception-specification* is needed for a defaulted special member function that calls the function. [*Note:* A defaulted declaration does not require the *exception-specification* of a base member function to be evaluated until the implicit *exception-specification* of the derived function is needed, but an explicit *exception-specification* needs the implicit *exception-specification* to compare against. — *end note*]

The *exception-specification* of a defaulted special member function is evaluated as described above only when needed; similarly, the *exception-specification* of a specialization of a function template or member function of a class template is instantiated only when needed.

3 In a *dynamic-exception-specification*, a *type-id* followed by an ellipsis is a pack expansion (14.703.1).

4

[*Note:* The use of *dynamic-exception-specification*s is deprecated (see Annex D). — *end note*]

15.308 Special functions [except.special]

1 The functions `std::terminate()` (15.308.1) and `std::unexpected()` (15.370.1) are used by the exception handling mechanism for coping with errors related to the exception handling mechanism itself. The function `std::current_exception()` (18.254.1) and the class `std::nested_exception` (18.265.1) can be used by a program to capture the currently handled exception.

15.308.1 The `std::terminate()` function [except.terminate]

1

In some situations exception handling must be abandoned for less subtle error handling techniques. [*Note:* These situations are:

— when the exception handling mechanism, after completing the initialization of the exception object but before activation of a handler for the exception (15.7), calls a function that exits via an exception, or

— when the exception handling mechanism cannot find a handler for a thrown exception (15.146), or

— when the search for a handler (15.146) encounters the outermost block of a function with a *noexcept-specification* that does not allow the exception (15.228), or

— when the destruction of an object during stack unwinding (15.89) terminates by throwing an exception, or

— when initialization of a non-local variable with static or thread storage duration (3.1937.1) exits via an exception, or

— when destruction of an object with static or thread storage duration exits via an exception (3.2200.1), or

— when execution of a function registered with `std::atexit` or `std::at_quick_exit` exits via an exception (18.127), or

— when a *throw-expression* with no operand attempts to rethrow an exception and no exception is being handled (15.7), or

— when `std::unexpected` throws an exception which is not allowed by the previously violated *dynamic-exception-specification*, and `std::bad_exception` is not included in that *dynamic-exception-specification* (15.370.1), or

— when the implementation's default unexpected exception handler is called (D.62.1), or

— when the function `std::nested_exception::rethrow_nested` is called for an object that has captured no exception (18.265.1), or

— when execution of the initial function of a thread exits via an exception (30.31.0.1), or

— when the destructor or the copy assignment operator is invoked on an object of type `std::thread` that refers to a joinable thread (30.60.0.1, 30.74.0.1).

— *end note*]

1

In such cases, `std::terminate()` is called (18.247.2). In the situation where no matching handler is found, it is implementation-defined whether or not the stack is unwound before `std::terminate()` is called. In the situation where the search for a handler (15.146) encounters the outermost block of a function with a *noexcept-specification* that does not allow the exception (15.228), it is implementation-defined whether the stack is unwound, unwound partially, or not unwound at all before `std::terminate()` is called. In all other situations, the stack shall not be unwound before `std::terminate()` is called. An implementation is not permitted to finish stack unwinding prematurely based on a determination that the unwind process will eventually cause a call to `std::terminate()`.

15.370.1 The `std::unexpected()` function [except.unexpected]

1 If a function with a *dynamic-exception-specification* throws an exception that is not listed in the *dynamic-exception-specification*, the function `std::unexpected()` is called (D.62) immediately after completing the stack unwinding for the former function.

2 [*Note:* By default, `std::unexpected()` calls `std::terminate()`, but a program can install its own handler function (D.62.2). In either case, the constraints in the following paragraph apply. — *end note*]

3

The `std::unexpected()` function shall not return, but it can throw (or re-throw) an exception. If it throws a new exception which is allowed by the exception specification which previously was violated, then the search for another handler will continue at the call of the function whose exception specification was violated. If it throws or rethrows an exception that the *dynamic-exception-specification* does not allow then the following happens: If the *dynamic-exception-specification* does not include the class `std::bad_exception` (18.247.1) then the function `std::terminate()` is called, otherwise the thrown exception is replaced by an implementation-defined object of the type `std::bad_exception` and the search for another handler will continue at the call of the function whose *dynamic-exception-specification* was violated.

1 Thus, a *dynamic-exception-specification* guarantees that only the listed exceptions will be thrown. If the *dynamic-exception-specification* includes the type `std::bad_exception` then any exception not on the list may be replaced by `std::bad_exception` within the function `std::unexpected()`.

15.371.1 The `std::uncaught_exception()` function [except.uncaught]

1

The function `std::uncaught_exception()` returns `true` after completing the initialization of the exception object (15.7) until completing the activation of a handler for the exception (15.146, 18.253.1). This includes

stack unwinding. If the exception is rethrown (15.7), `std::uncaught_exception()` returns `true` from the point of rethrow until the rethrown exception is caught again.

16 Preprocessing directives [cpp]

1

A *preprocessing directive* consists of a sequence of preprocessing tokens that satisfies the following constraints: The first token in the sequence is a `#` preprocessing token that (at the start of translation phase 4) is either the first character in the source file (optionally after white space containing no new-line characters) or that follows white space containing at least one new-line character. The last token in the sequence is the first new-line character that follows the first token in the sequence.[146] A new-line character ends the preprocessing directive even if it occurs within what would otherwise be an invocation of a function-like macro.

preprocessing-file:
  group$_{opt}$

group:
  group-part
  group group-part

group-part:
  if-section
  control-line
  text-line
  `#` *non-directive*

if-section:
  if-group elif-groups$_{opt}$ *else-group*$_{opt}$ *endif-line*

if-group:
  `# if` *constant-expression new-line group*$_{opt}$
  `# ifdef` *identifier new-line group*$_{opt}$
  `# ifndef` *identifier new-line group*$_{opt}$

elif-groups:
  elif-group
  elif-groups elif-group

elif-group:
  `# elif` *constant-expression new-line group*$_{opt}$

else-group:
  `# else` *new-line group*$_{opt}$

endif-line:
  `# endif` *new-line*

146) Thus, preprocessing directives are commonly called "lines." These "lines" have no other syntactic significance, as all white space is equivalent except in certain situations during preprocessing (see the `#` character string literal creation operator in 16.161.1, for example).

control-line:
`# include` *pp-tokens new-line*
`# define` *identifier replacement-list new-line*
`# define` *identifier lparen identifier-list*$_{opt}$ `)` *replacement-list new-line*
`# define` *identifier lparen* `...` `)` *replacement-list new-line*
`# define` *identifier lparen identifier-list,* `...` `)` *replacement-list new-line*
`# undef` *identifier new-line*
`# line` *pp-tokens new-line*
`# error` *pp-tokens*$_{opt}$ *new-line*
`# pragma` *pp-tokens*$_{opt}$ *new-line*
`#` *new-line*

text-line:
pp-tokens$_{opt}$ *new-line*

non-directive:
pp-tokens new-line

lparen:
a `(` character not immediately preceded by white-space

identifier-list:
identifier
identifier-list `,` *identifier*

replacement-list:
pp-tokens$_{opt}$

pp-tokens:
preprocessing-token
pp-tokens preprocessing-token

new-line:
the new-line character

1

A text line shall not begin with a `#` preprocessing token. A non-directive shall not begin with any of the directive names appearing in the syntax.

1 When in a group that is skipped (16.18), the directive syntax is relaxed to allow any sequence of preprocessing tokens to occur between the directive name and the following new-line character.

2

The only white-space characters that shall appear between preprocessing tokens within a preprocessing directive (from just after the introducing `#` preprocessing token through just before the terminating new-line character) are space and horizontal-tab (including spaces that have replaced comments or possibly other white-space characters in translation phase 3).

1

The implementation can process and skip sections of source files conditionally, include other source files, and replace macros. These capabilities are called *preprocessing*, because conceptually they occur before translation of the resulting translation unit.

1

The preprocessing tokens within a preprocessing directive are not subject to macro expansion unless otherwise stated.

[*Example:* In:

```
#define EMPTY
EMPTY   #   include <file.h>
```

the sequence of preprocessing tokens on the second line is *not* a preprocessing directive, because it does not begin with a # at the start of translation phase 4, even though it will do so after the macro `EMPTY` has been replaced. — *end example*]

16.18 Conditional inclusion [cpp.cond]

1

The expression that controls conditional inclusion shall be an integral constant expression except that identifiers (including those lexically identical to keywords) are interpreted as described below[147] and it may contain unary operator expressions of the form

`defined` *identifier*

or

`defined (` *identifier* `)`

which evaluate to `1` if the identifier is currently defined as a macro name (that is, if it is predefined or if it has been the subject of a `#define` preprocessing directive without an intervening `#undef` directive with the same subject identifier), `0` if it is not.

1 Each preprocessing token that remains (in the list of preprocessing tokens that will become the controlling expression) after all macro replacements have occurred shall be in the lexical form of a token (2.247).

2 Preprocessing directives of the forms

`# if` *constant-expression new-line* $group_{opt}$
`# elif` *constant-expression new-line* $group_{opt}$

check whether the controlling constant expression evaluates to nonzero.

3

Prior to evaluation, macro invocations in the list of preprocessing tokens that will become the controlling constant expression are replaced (except for those macro names modified by the `defined` unary operator), just as in normal text. If the token `defined` is generated as a result of this replacement process or use of the `defined` unary operator does not match one of the two specified forms prior to macro replacement, the behavior is undefined. After all replacements due to macro expansion and the `defined` unary operator have been performed, all remaining identifiers and keywords[148], except for `true` and `false`, are replaced with the pp-number `0`, and then each preprocessing token is converted into a token. The resulting tokens comprise the controlling constant expression which is evaluated according to the rules of 5.3643 using arithmetic that has at least the ranges specified in 18.41. For the purposes of this token conversion and evaluation all signed and unsigned integer types act as if they have the same representation as, respectively, `intmax_t` or `uintmax_t` (18.114).[149] This includes interpreting character literals, which may involve converting escape sequences into execution character set members. Whether the numeric value for these character literals matches the value obtained when an identical character literal occurs in an expression (other than within a `#if` or `#elif` directive) is implementation-defined.[150] Also, whether a single-character character literal may

147) Because the controlling constant expression is evaluated during translation phase 4, all identifiers either are or are not macro names — there simply are no keywords, enumeration constants, etc.

148) An alternative token (2.236) is not an identifier, even when its spelling consists entirely of letters and underscores. Therefore it is not subject to this replacement.

149) Thus on an implementation where `std::numeric_limits<int>::max()` is `0x7FFF` and `std::numeric_limits<unsigned int>::max()` is `0xFFFF`, the integer literal `0x8000` is signed and positive within a `#if` expression even though it is unsigned in translation phase 7 (2.6).

150) Thus, the constant expression in the following `#if` directive and `if` statement is not guaranteed to evaluate to the same value in these two contexts.

```
#if 'z' - 'a' == 25
if ('z' - 'a' == 25)
```

have a negative value is implementation-defined. Each subexpression with type `bool` is subjected to integral promotion before processing continues.

1 Preprocessing directives of the forms

`# ifdef` *identifier new-line* $group_{opt}$
`# ifndef` *identifier new-line* $group_{opt}$

check whether the identifier is or is not currently defined as a macro name. Their conditions are equivalent to `#if defined` *identifier* and `#if !defined` *identifier* respectively.

2

Each directive's condition is checked in order. If it evaluates to false (zero), the group that it controls is skipped: directives are processed only through the name that determines the directive in order to keep track of the level of nested conditionals; the rest of the directives' preprocessing tokens are ignored, as are the other preprocessing tokens in the group. Only the first group whose control condition evaluates to true (nonzero) is processed. If none of the conditions evaluates to true, and there is a `#else` directive, the group controlled by the `#else` is processed; lacking a `#else` directive, all the groups until the `#endif` are skipped.[151]

16.48 Source file inclusion [cpp.include]

1

A `#include` directive shall identify a header or source file that can be processed by the implementation.

1

A preprocessing directive of the form

`# include <` *h-char-sequence*`>` *new-line*

searches a sequence of implementation-defined places for a header identified uniquely by the specified sequence between the `<` and `>` delimiters, and causes the replacement of that directive by the entire contents of the header. How the places are specified or the header identified is implementation-defined.

1

A preprocessing directive of the form

`# include "` *q-char-sequence*`"` *new-line*

causes the replacement of that directive by the entire contents of the source file identified by the specified sequence between the `"` delimiters. The named source file is searched for in an implementation-defined manner. If this search is not supported, or if the search fails, the directive is reprocessed as if it read

`# include <` *h-char-sequence*`>` *new-line*

with the identical contained sequence (including `>` characters, if any) from the original directive.

1

A preprocessing directive of the form

`# include` *pp-tokens new-line*

(that does not match one of the two previous forms) is permitted. The preprocessing tokens after `include` in the directive are processed just as in normal text (i.e., each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens). If the directive resulting after all replacements does not match one of the two previous forms, the behavior is undefined.[152] The method by which a sequence

151) As indicated by the syntax, a preprocessing token shall not follow a `#else` or `#endif` directive before the terminating new-line character. However, comments may appear anywhere in a source file, including within a preprocessing directive.
152) Note that adjacent string literals are not concatenated into a single string literal (see the translation phases in 2.6); thus, an expansion that results in two string literals is an invalid directive.

of preprocessing tokens between a `<` and a `>` preprocessing token pair or a pair of `"` characters is combined into a single header name preprocessing token is implementation-defined.

1

The implementation shall provide unique mappings for sequences consisting of one or more *nondigit*s or *digit*s (2.268) followed by a period (`.`) and a single *nondigit*. The first character shall not be a *digit*. The implementation may ignore distinctions of alphabetical case.

1

A `#include` preprocessing directive may appear in a source file that has been read because of a `#include` directive in another file, up to an implementation-defined nesting limit.

1 [*Note:* Although an implementation may provide a mechanism for making arbitrary source files available to the `< >` search, in general programmers should use the `< >` form for headers provided with the implementation, and the `" "` form for sources outside the control of the implementation. For instance:

```
#include <stdio.h>
#include <unistd.h>
#include "usefullib.h"
#include "myprog.h"
```

— *end note*]

2 [*Example:* This illustrates macro-replaced `#include` directives:

```
#if VERSION == 1
    #define INCFILE  "vers1.h"
#elif VERSION == 2
    #define INCFILE  "vers2.h"   // and so on
#else
    #define INCFILE  "versN.h"
#endif
#include INCFILE
```

— *end example*]

16.116 Macro replacement [cpp.replace]

1 Two replacement lists are identical if and only if the preprocessing tokens in both have the same number, ordering, spelling, and white-space separation, where all white-space separations are considered identical.

2

An identifier currently defined as an *object-like* macro may be redefined by another `#define` preprocessing directive provided that the second definition is an object-like macro definition and the two replacement lists are identical, otherwise the program is ill-formed. Likewise, an identifier currently defined as a *function-like* macro may be redefined by another `#define` preprocessing directive provided that the second definition is a function-like macro definition that has the same number and spelling of parameters, and the two replacement lists are identical, otherwise the program is ill-formed.

1

There shall be white-space between the identifier and the replacement list in the definition of an object-like macro.

1

If the identifier-list in the macro definition does not end with an ellipsis, the number of arguments (including those arguments consisting of no preprocessing tokens) in an invocation of a function-like macro shall equal the number of parameters in the macro definition. Otherwise, there shall be more arguments in the invocation

than there are parameters in the macro definition (excluding the `...`). There shall exist a `)` preprocessing token that terminates the invocation.

1 The identifier `__VA_ARGS__` shall occur only in the replacement-list of a function-like macro that uses the ellipsis notation in the parameters.

2 A parameter identifier in a function-like macro shall be uniquely declared within its scope.

3 The identifier immediately following the `define` is called the *macro name*. There is one name space for macro names. Any white-space characters preceding or following the replacement list of preprocessing tokens are not considered part of the replacement list for either form of macro.

4

If a `#` preprocessing token, followed by an identifier, occurs lexically at the point at which a preprocessing directive could begin, the identifier is not subject to macro replacement.

1

A preprocessing directive of the form

`#` `define` *identifier replacement-list new-line*

defines an *object-like macro* that causes each subsequent instance of the macro name[153] to be replaced by the replacement list of preprocessing tokens that constitute the remainder of the directive.[154] The replacement list is then rescanned for more macro names as specified below.

1

A preprocessing directive of the form

`#` `define` *identifier lparen identifier-list*$_{opt}$`)` *replacement-list new-line*
`#` `define` *identifier lparen* `...` `)` *replacement-list new-line*
`#` `define` *identifier lparen identifier-list* `,` `...` `)` *replacement-list new-line*

defines a *function-like macro* with parameters, whose use is similar syntactically to a function call. The parameters are specified by the optional list of identifiers, whose scope extends from their declaration in the identifier list until the new-line character that terminates the `#define` preprocessing directive. Each subsequent instance of the function-like macro name followed by a `(` as the next preprocessing token introduces the sequence of preprocessing tokens that is replaced by the replacement list in the definition (an invocation of the macro). The replaced sequence of preprocessing tokens is terminated by the matching `)` preprocessing token, skipping intervening matched pairs of left and right parenthesis preprocessing tokens. Within the sequence of preprocessing tokens making up an invocation of a function-like macro, new-line is considered a normal white-space character.

1

The sequence of preprocessing tokens bounded by the outside-most matching parentheses forms the list of arguments for the function-like macro. The individual arguments within the list are separated by comma preprocessing tokens, but comma preprocessing tokens between matching inner parentheses do not separate arguments. If there are sequences of preprocessing tokens within the list of arguments that would otherwise act as preprocessing directives,[155] the behavior is undefined.

1

If there is a `...` immediately preceding the `)` in the function-like macro definition, then the trailing arguments, including any separating comma preprocessing tokens, are merged to form a single item: the *variable*

153) Since, by macro-replacement time, all character literals and string literals are preprocessing tokens, not sequences possibly containing identifier-like subsequences (see 2.6, translation phases), they are never scanned for macro names or parameters.
154) An alternative token (2.236) is not an identifier, even when its spelling consists entirely of letters and underscores. Therefore it is not possible to define a macro whose name is the same as that of an alternative token.
155) Despite the name, a non-directive is a preprocessing directive.

arguments. The number of arguments so combined is such that, following merger, the number of arguments is one more than the number of parameters in the macro definition (excluding the `...`).

16.146.1 Argument substitution [cpp.subst]

1

After the arguments for the invocation of a function-like macro have been identified, argument substitution takes place. A parameter in the replacement list, unless preceded by a `#` or `##` preprocessing token or followed by a `##` preprocessing token (see below), is replaced by the corresponding argument after all macros contained therein have been expanded. Before being substituted, each argument's preprocessing tokens are completely macro replaced as if they formed the rest of the preprocessing file; no other preprocessing tokens are available.

1

An identifier `__VA_ARGS__` that occurs in the replacement list shall be treated as if it were a parameter, and the variable arguments shall form the preprocessing tokens used to replace it.

16.161.1 The # operator [cpp.stringize]

1 Each `#` preprocessing token in the replacement list for a function-like macro shall be followed by a parameter as the next preprocessing token in the replacement list.

2

A *character string literal* is a *string-literal* with no prefix. If, in the replacement list, a parameter is immediately preceded by a `#` preprocessing token, both are replaced by a single character string literal preprocessing token that contains the spelling of the preprocessing token sequence for the corresponding argument. Each occurrence of white space between the argument's preprocessing tokens becomes a single space character in the character string literal. White space before the first preprocessing token and after the last preprocessing token comprising the argument is deleted. Otherwise, the original spelling of each preprocessing token in the argument is retained in the character string literal, except for special handling for producing the spelling of string literals and character literals: a `\` character is inserted before each `"` and `\` character of a character literal or string literal (including the delimiting `"` characters). If the replacement that results is not a valid character string literal, the behavior is undefined. The character string literal corresponding to an empty argument is `""`. The order of evaluation of `#` and `##` operators is unspecified.

16.175.1 The ## operator [cpp.concat]

1

A `##` preprocessing token shall not occur at the beginning or at the end of a replacement list for either form of macro definition.

1

If, in the replacement list of a function-like macro, a parameter is immediately preceded or followed by a `##` preprocessing token, the parameter is replaced by the corresponding argument's preprocessing token sequence; however, if an argument consists of no preprocessing tokens, the parameter is replaced by a placemarker preprocessing token instead.[156]

1

For both object-like and function-like macro invocations, before the replacement list is reexamined for more macro names to replace, each instance of a `##` preprocessing token in the replacement list (not from an argument) is deleted and the preceding preprocessing token is concatenated with the following preprocessing

156) Placemarker preprocessing tokens do not appear in the syntax because they are temporary entities that exist only within translation phase 4.

token. Placemarker preprocessing tokens are handled specially: concatenation of two placemarkers results in a single placemarker preprocessing token, and concatenation of a placemarker with a non-placemarker preprocessing token results in the non-placemarker preprocessing token. If the result is not a valid preprocessing token, the behavior is undefined. The resulting token is available for further macro replacement. The order of evaluation of `##` operators is unspecified.

[*Example:* In the following fragment:

```
#define hash_hash # ## #
#define mkstr(a) # a
#define in_between(a) mkstr(a)
#define join(c, d) in_between(c hash_hash d)
char p[] = join(x, y);          // equivalent to
                                // char p[] = "x ## y";
```

The expansion produces, at various stages:

```
join(x, y)
in_between(x hash_hash y)
in_between(x ## y)
mkstr(x ## y)
"x ## y"
```

In other words, expanding `hash_hash` produces a new token, consisting of two adjacent sharp signs, but this new token is not the `##` operator. — *end example*]

16.187.1 Rescanning and further replacement [cpp.rescan]

1

After all parameters in the replacement list have been substituted and `#` and `##` processing has taken place, all placemarker preprocessing tokens are removed. Then the resulting preprocessing token sequence is rescanned, along with all subsequent preprocessing tokens of the source file, for more macro names to replace.

1

If the name of the macro being replaced is found during this scan of the replacement list (not including the rest of the source file's preprocessing tokens), it is not replaced. Furthermore, if any nested replacements encounter the name of the macro being replaced, it is not replaced. These nonreplaced macro name preprocessing tokens are no longer available for further replacement even if they are later (re)examined in contexts in which that macro name preprocessing token would otherwise have been replaced.

1

The resulting completely macro-replaced preprocessing token sequence is not processed as a preprocessing directive even if it resembles one, but all pragma unary operator expressions within it are then processed as specified in 16.276 below.

16.220.1 Scope of macro definitions [cpp.scope]

1 A macro definition lasts (independent of block structure) until a corresponding `#undef` directive is encountered or (if none is encountered) until the end of the translation unit. Macro definitions have no significance after translation phase 4.

2

A preprocessing directive of the form

`# undef` *identifier new-line*

causes the specified identifier no longer to be defined as a macro name. It is ignored if the specified identifier is not currently defined as a macro name.

1 [*Example:* The simplest use of this facility is to define a "manifest constant," as in

```
#define TABSIZE 100
int table[TABSIZE];
```

— *end example*]

2 [*Example:* The following defines a function-like macro whose value is the maximum of its arguments. It has the advantages of working for any compatible types of the arguments and of generating in-line code without the overhead of function calling. It has the disadvantages of evaluating one or the other of its arguments a second time (including side effects) and generating more code than a function if invoked several times. It also cannot have its address taken, as it has none.

```
#define max(a, b) ((a) > (b) ? (a) : (b))
```

The parentheses ensure that the arguments and the resulting expression are bound properly. — *end example*]

3 [*Example:* To illustrate the rules for redefinition and reexamination, the sequence

```
#define x       3
#define f(a)    f(x * (a))
#undef  x
#define x       2
#define g       f
#define z       z[0]
#define h       g(~
#define m(a)    a(w)
#define w       0,1
#define t(a)    a
#define p()     int
#define q(x)    x
#define r(x,y)  x ## y
#define str(x)  # x

f(y+1) + f(f(z)) % t(t(g)(0) + t)(1);
g(x+(3,4)-w) | h 5) & m
    (f)^m(m);
p() i[q()] = { q(1), r(2,3), r(4,), r(,5), r(,) };
char c[2][6] = { str(hello), str() };
```

results in

```
f(2 * (y+1)) + f(2 * (f(2 * (z[0])))) % f(2 * (0)) + t(1);
f(2 * (2+(3,4)-0,1)) | f(2 * (~ 5)) & f(2 * (0,1))^m(0,1);
int i[] = { 1, 23, 4, 5, };
char c[2][6] = { "hello", "" };
```

— *end example*]

4 [*Example:* To illustrate the rules for creating character string literals and concatenating tokens, the sequence

```
#define str(s)      # s
#define xstr(s)     str(s)
#define debug(s, t) printf("x" # s "= %d, x" # t "= %s", \
                x ## s, x ## t)
#define INCFILE(n)  vers ## n
```

```
#define glue(a, b)  a ## b
#define xglue(a, b) glue(a, b)
#define HIGHLOW     "hello"
#define LOW         LOW ", world"

debug(1, 2);
fputs(str(strncmp("abc\0d", "abc", '\4')  // this goes away
    == 0) str(: @\n), s);
#include xstr(INCFILE(2).h)
glue(HIGH, LOW);
xglue(HIGH, LOW)
```

results in

```
printf("x" "1" "= %d, x" "2" "= %s", x1, x2);
fputs("strncmp(\"abc\\0d\", \"abc\", '\\4') == 0" ": @\n", s);
#include "vers2.h"    (after macro replacement, before file access)
"hello";
"hello" ", world"
```

or, after concatenation of the character string literals,

```
printf("x1= %d, x2= %s", x1, x2);
fputs("strncmp(\"abc\\0d\", \"abc\", '\\4') == 0: @\n", s);
#include "vers2.h"    (after macro replacement, before file access)
"hello";
"hello, world"
```

Space around the # and ## tokens in the macro definition is optional. — *end example*]

5 [*Example:* To illustrate the rules for placemarker preprocessing tokens, the sequence

```
#define t(x,y,z) x ## y ## z
int j[] = { t(1,2,3), t(,4,5), t(6,,7), t(8,9,),
  t(10,,), t(,11,), t(,,12), t(,,) };
```

results in

```
int j[] = { 123, 45, 67, 89,
  10, 11, 12, };
```

— *end example*]

6 [*Example:* To demonstrate the redefinition rules, the following sequence is valid.

```
#define OBJ_LIKE      (1-1)
#define OBJ_LIKE      /* white space */ (1-1) /* other */
#define FUNC_LIKE(a)   ( a )
#define FUNC_LIKE( a )(     /* note the white space */ \
                a /* other stuff on this line
                  */ )
```

But the following redefinitions are invalid:

```
#define OBJ_LIKE    (0)       // different token sequence
#define OBJ_LIKE    (1 - 1)   // different white space
#define FUNC_LIKE(b) ( a )    // different parameter usage
#define FUNC_LIKE(b) ( b )    // different parameter spelling
```

— *end example*]

7 [*Example:* Finally, to show the variable argument list macro facilities:

```
#define debug(...) fprintf(stderr, __VA_ARGS__)
#define showlist(...) puts(#__VA_ARGS__)
#define report(test, ...) ((test) ? puts(#test) : printf(__VA_ARGS__))
debug("Flag");
debug("X = %d\n", x);
showlist(The first, second, and third items.);
report(x>y, "x is %d but y is %d", x, y);
```

results in

```
fprintf(stderr, "Flag");
fprintf(stderr, "X = %d\n", x);
puts("The first, second, and third items.");
((x>y) ? puts("x>y") : printf("x is %d but y is %d", x, y));
```

— *end example*]

16.222 Line control [cpp.line]

1 The string literal of a `#line` directive, if present, shall be a character string literal.

2 The *line number* of the current source line is one greater than the number of new-line characters read or introduced in translation phase 1 (2.6) while processing the source file to the current token.

3

A preprocessing directive of the form

`# line` *digit-sequence new-line*

causes the implementation to behave as if the following sequence of source lines begins with a source line that has a line number as specified by the digit sequence (interpreted as a decimal integer). If the digit sequence specifies zero or a number greater than 2147483647, the behavior is undefined.

1

A preprocessing directive of the form

`# line` *digit-sequence* `"` *s-char-sequence*$_{opt}$`"` *new-line*

sets the presumed line number similarly and changes the presumed name of the source file to be the contents of the character string literal.

1

A preprocessing directive of the form

`# line` *pp-tokens new-line*

(that does not match one of the two previous forms) is permitted. The preprocessing tokens after `line` on the directive are processed just as in normal text (each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens). If the directive resulting after all replacements does not match one of the two previous forms, the behavior is undefined; otherwise, the result is processed as appropriate.

16.226 Error directive [cpp.error]

1

A preprocessing directive of the form

`# error` *pp-tokens*$_{opt}$ *new-line*

causes the implementation to produce a diagnostic message that includes the specified sequence of preprocessing tokens, and renders the program ill-formed.

16.228 Pragma directive [cpp.pragma]

1

A preprocessing directive of the form

`# pragma` *pp-tokens*$_{opt}$ *new-line*

causes the implementation to behave in an implementation-defined manner. The behavior might cause translation to fail or cause the translator or the resulting program to behave in a non-conforming manner. Any pragma that is not recognized by the implementation is ignored.

16.237 Null directive [cpp.null]

1 A preprocessing directive of the form

`#` *new-line*

has no effect.

16.238 Predefined macro names [cpp.predefined]

1

The following macro names shall be defined by the implementation:

`__cplusplus`
: The name `__cplusplus` is defined to the value `201402L` when compiling a C++ translation unit.[157]

`__DATE__`
: The date of translation of the source file: a character string literal of the form `"Mmm dd yyyy"`, where the names of the months are the same as those generated by the `asctime` function, and the first character of `dd` is a space character if the value is less than 10. If the date of translation is not available, an implementation-defined valid date shall be supplied.

`__FILE__`
: The presumed name of the current source file (a character string literal).[158]

`__LINE__`
: The presumed line number (within the current source file) of the current source line (an integer literal).[158]

`__STDC_HOSTED__`
: The integer literal `1` if the implementation is a hosted implementation or the integer literal `0` if it is not.

`__TIME__`
: The time of translation of the source file: a character string literal of the form `"hh:mm:ss"` as in the time generated by the `asctime` function. If the time of translation is not available, an implementation-defined valid time shall be supplied.

1

The following macro names are conditionally defined by the implementation:

157) It is intended that future versions of this standard will replace the value of this macro with a greater value. Non-conforming compilers should use a value with at most five decimal digits.
158) The presumed source file name and line number can be changed by the `#line` directive.

`__STDC__`
Whether `__STDC__` is predefined and if so, what its value is, are implementation-defined.

`__STDC_MB_MIGHT_NEQ_WC__`
The integer literal `1`, intended to indicate that, in the encoding for `wchar_t`, a member of the basic character set need not have a code value equal to its value when used as the lone character in an ordinary character literal.

`__STDC_VERSION__`
Whether `__STDC_VERSION__` is predefined and if so, what its value is, are implementation-defined.

`__STDC_ISO_10646__`
An integer literal of the form `yyyymmL` (for example, `199712L`). If this symbol is defined, then every character in the Unicode required set, when stored in an object of type `wchar_t`, has the same value as the short identifier of that character. The *Unicode required set* consists of all the characters that are defined by ISO/IEC 10646, along with all amendments and technical corrigenda as of the specified year and month.

`__STDCPP_STRICT_POINTER_SAFETY__`
Defined, and has the value integer literal 1, if and only if the implementation has strict pointer safety (3.2532.0.1).

`__STDCPP_THREADS__`
Defined, and has the value integer literal 1, if and only if a program can have more than one thread of execution (1.859).

1 The values of the predefined macros (except for `__FILE__` and `__LINE__`) remain constant throughout the translation unit.

2 If any of the pre-defined macro names in this subclause, or the identifier `defined`, is the subject of a `#define` or a `#undef` preprocessing directive, the behavior is undefined. Any other predefined macro names shall begin with a leading underscore followed by an uppercase letter or a second underscore.

16.276 Pragma operator [cpp.pragma.op]

A unary operator expression of the form:

`_Pragma (` *string-literal* `)`

is processed as follows: The string literal is *destringized* by deleting the `L` prefix, if present, deleting the leading and trailing double-quotes, replacing each escape sequence `\"` by a double-quote, and replacing each escape sequence `\\` by a single backslash. The resulting sequence of characters is processed through translation phase 3 to produce preprocessing tokens that are executed as if they were the *pp-tokens* in a pragma directive. The original four preprocessing tokens in the unary operator expression are removed.

[*Example:*

```
#pragma listing on "..\listing.dir"
```

can also be expressed as:

```
_Pragma ( "listing on \"..\\listing.dir\"" )
```

The latter form is processed in the same way whether it appears literally as shown, or results from macro replacement, as in:

```
#define LISTING(x) PRAGMA(listing on #x)
#define PRAGMA(x) _Pragma(#x)

LISTING( ..\listing.dir )
```

— *end example*]

17 Library introduction [library]

17.1 General [library.general]

1 This Clause describes the contents of the *C++ standard library*, how a well-formed C++ program makes use of the library, and how a conforming implementation may provide the entities in the library.

2

The following subclauses describe the definitions (17.9), method of description (17.11), and organization (17.37.1) of the library. Clause 17.37, Clauses 18 through 30, and Annex D specify the contents of the library, as well as library requirements and constraints on both well-formed C++ programs and conforming implementations.

1 Detailed specifications for each of the components in the library are in Clauses 18–30, as shown in Table 13.

Table 13 — Library categories

Clause	Category
18	Language support library
19	Diagnostics library
20	General utilities library
21	Strings library
22	Localization library
23	Containers library
24	Iterators library
25	Algorithms library
26	Numerics library
27	Input/output library
28	Regular expressions library
29	Atomic operations library
30	Thread support library

2 The language support library (Clause 18) provides components that are required by certain parts of the C++ language, such as memory allocation (5.2221.1, 5.2455.1) and exception processing (Clause 15).

3 The diagnostics library (Clause 19) provides a consistent framework for reporting errors in a C++ program, including predefined exception classes.

4 The general utilities library (Clause 20) includes components used by other library elements, such as a predefined storage allocator for dynamic storage management (3.2331.1), and components used as infrastructure in C++ programs, such as a tuples, function wrappers, and time facilities.

5 The strings library (Clause 21) provides support for manipulating text represented as sequences of type `char`, sequences of type `char16_t`, sequences of type `char32_t`, sequences of type `wchar_t`, and sequences of any other character-like type.

6 The localization library (Clause 22) provides extended internationalization support for text processing.

7 The containers (Clause 23), iterators (Clause 24), and algorithms (Clause 25) libraries provide a C++ program with access to a subset of the most widely used algorithms and data structures.

8 The numerics library (Clause 26) provides numeric algorithms and complex number components that extend support for numeric processing. The `valarray` component provides support for n-at-a-time processing, potentially implemented as parallel operations on platforms that support such processing. The random

number component provides facilities for generating pseudo-random numbers.

9 The input/output library (Clause 27) provides the `iostream` components that are the primary mechanism for C++ program input and output. They can be used with other elements of the library, particularly strings, locales, and iterators.

10 The regular expressions library (Clause 28) provides regular expression matching and searching.

11 The atomic operations library (Clause 29) allows more fine-grained concurrent access to shared data than is possible with locks.

12 The thread support library (Clause 30) provides components to create and manage threads, including mutual exclusion and interthread communication.

17.3 The C standard library [library.c]

1

The C++ standard library also makes available the facilities of the C standard library, suitably adjusted to ensure static type safety.

1

The descriptions of many library functions rely on the C standard library for the signatures and semantics of those functions. In all such cases, any use of the `restrict` qualifier shall be omitted.

17.9 Definitions [definitions]

17.9.1 [defns.arbitrary.stream]

arbitrary-positional stream

a stream (described in Clause 27) that can seek to any integral position within the length of the stream

[*Note:* Every arbitrary-positional stream is also a repositional stream. — *end note*]

17.9.2 [defns.block]

block

place a thread in the blocked state

17.9.3 [defns.blocked]

blocked thread

a thread that is waiting for some condition (other than the availability of a processor) to be satisfied before it can continue execution[159]

17.9.4 [defns.character]

character

<Clauses 21, 22, 27, and 28> any object which, when treated sequentially, can represent text

[*Note:* The term does not mean only `char`, `char16_t`, `char32_t`, and `wchar_t` objects, but any value that can be represented by a type that provides the definitions specified in these Clauses. — *end note*]

17.9.5 [defns.character.container]

character container type

a class or a type used to represent a *character*

[*Note:* It is used for one of the template parameters of the string, iostream, and regular expression class templates. A character container type is a POD (3.2955) type. — *end note*]

159) This definition is taken from POSIX.

17.9.6 **[defns.comparison]**

comparison function

an operator function (13.1122) for any of the equality (5.3247) or relational (5.3203) operators

17.9.7 **[defns.component]**

component

a group of library entities directly related as members, parameters, or return types

[*Note:* For example, the class template `basic_string` and the non-member function templates that operate on strings are referred to as the *string component*. — *end note*]

17.9.8 **[defns.deadlock]**

deadlock

one or more threads are unable to continue execution because each is blocked waiting for one or more of the others to satisfy some condition

17.9.9 **[defns.default.behavior.impl]**

default behavior

<implementation> any specific behavior provided by the implementation, within the scope of the *required behavior*

17.9.10 **[defns.default.behavior.func]**

default behavior

<specification> a description of *replacement function* and *handler function* semantics

17.9.11 **[defns.handler]**

handler function

a *non-reserved function* whose definition may be provided by a C++ program

[*Note:* A C++ program may designate a handler function at various points in its execution by supplying a pointer to the function when calling any of the library functions that install handler functions (Clause 18). — *end note*]

17.9.12 **[defns.iostream.templates]**

iostream class templates

templates, defined in Clause 27, that take two template arguments

[*Note:* The arguments are named `charT` and `traits`. The argument `charT` is a character container class, and the argument `traits` is a class which defines additional characteristics and functions of the character type represented by `charT` necessary to implement the iostream class templates. — *end note*]

17.9.13 **[defns.modifier]**

modifier function

a class member function (9.401) other than a constructor, assignment operator, or destructor that alters the state of an object of the class

17.9.14 **[defns.move.constr]**

move construction

direct-initialization of an object of some type with an rvalue of the same type

17.9.15 **[defns.move.assign]**

move assignment

assignment of an rvalue of some object type to a modifiable lvalue of the same type

17.9.16 **[defns.obj.state]**

object state

the current value of all non-static class members of an object (9.194)

[*Note:* The state of an object can be obtained by using one or more *observer functions*. — *end note*]

17.9.17 **[defns.ntcts]**

NTCTS

a sequence of values that have *character type* that precede the terminating null character type value `charT()`

17.9.18 **[defns.observer]**

observer function

a class member function (9.401) that accesses the state of an object of the class but does not alter that state

[*Note:* Observer functions are specified as `const` member functions (9.508.1). — *end note*]

17.9.19 **[defns.referenceable]**

referenceable type

An object type, a function type that does not have cv-qualifiers or a *ref-qualifier*, or a reference type. [*Note:* The term describes a type to which a reference can be created, including reference types. — *end note*]

17.9.20 **[defns.replacement]**

replacement function

a *non-reserved function* whose definition is provided by a C++ program

[*Note:* Only one definition for such a function is in effect for the duration of the program's execution, as the result of creating the program (2.6) and resolving the definitions of all translation units (3.1459). — *end note*]

17.9.21 **[defns.repositional.stream]**

repositional stream

a stream (described in Clause 27) that can seek to a position that was previously encountered

17.9.22 **[defns.required.behavior]**

required behavior

a description of *replacement function* and *handler function* semantics applicable to both the behavior provided by the implementation and the behavior of any such function definition in the program

[*Note:* If such a function defined in a C++ program fails to meet the required behavior when it executes, the behavior is undefined. — *end note*]

17.9.23 **[defns.reserved.function]**

reserved function

a function, specified as part of the C++ standard library, that must be defined by the implementation

[*Note:* If a C++ program provides a definition for any reserved function, the results are undefined. — *end note*]

17.9.24 **[defns.stable]**

stable algorithm

an algorithm that preserves, as appropriate to the particular algorithm, the order of elements

[*Note:* Requirements for stable algorithms are given in 17.602.0.1. — *end note*]

17.9.25 **[defns.traits]**

traits class

a class that encapsulates a set of types and functions necessary for class templates and function templates to manipulate objects of types for which they are instantiated

[*Note:* Traits classes defined in Clauses 21, 22 and 27 are *character traits*, which provide the character handling support needed by the string and iostream classes. — *end note*]

17.9.26 **[defns.unblock]**

unblock

place a thread in the unblocked state

17.9.27 **[defns.valid]**

valid but unspecified state

an object state that is not specified except that the object's invariants are met and operations on the object behave as specified for its type

[*Example:* If an object `x` of type `std::vector<int>` is in a valid but unspecified state, `x.empty()` can be called unconditionally, and `x.front()` can be called only if `x.empty()` returns `false`. — *end example*]

17.10 Additional definitions [defns.additional]

1 1.27 defines additional terms used elsewhere in this International Standard.

17.11 Method of description (Informative) [description]

1 This subclause describes the conventions used to specify the C++ standard library. 17.11.1 describes the structure of the normative Clauses 18 through 30 and Annex D. 17.17.1 describes other editorial conventions.

17.11.1 Structure of each clause [structure]

17.11.1.1 Elements [structure.elements]

1 Each library clause contains the following elements, as applicable:[160]

(1.1) — Summary

(1.2) — Requirements

(1.3) — Detailed specifications

(1.4) — References to the Standard C library

17.11.1.2 Summary [structure.summary]

1 The Summary provides a synopsis of the category, and introduces the first-level subclauses. Each subclause also provides a summary, listing the headers specified in the subclause and the library entities provided in each header.

160) To save space, items that do not apply to a Clause are omitted. For example, if a Clause does not specify any requirements, there will be no "Requirements" subclause.

2

Paragraphs labeled "Note(s):" or "Example(s):" are informative, other paragraphs are normative.

1 The contents of the summary and the detailed specifications include:

(1.1) — macros

(1.2) — values

(1.3) — types

(1.4) — classes and class templates

(1.5) — functions and function templates

(1.6) — objects

17.12.0.1 Requirements [structure.requirements]

1 Requirements describe constraints that shall be met by a C++ program that extends the standard library. Such extensions are generally one of the following:

(1.1) — Template arguments

(1.2) — Derived classes

(1.3) — Containers, iterators, and algorithms that meet an interface convention

2 The string and iostream components use an explicit representation of operations required of template arguments. They use a class template `char_traits` to define these constraints.

3

Interface convention requirements are stated as generally as possible. Instead of stating "class X has to define a member function `operator++()`," the interface requires "for any object `x` of class `X`, `++x` is defined." That is, whether the operator is a member is unspecified.

1 Requirements are stated in terms of well-defined expressions that define valid terms of the types that satisfy the requirements. For every set of well-defined expression requirements there is a table that specifies an initial set of the valid expressions and their semantics. Any generic algorithm (Clause 25) that uses the well-defined expression requirements is described in terms of the valid expressions for its template type parameters.

2 Template argument requirements are sometimes referenced by name. See 17.17.1.1.

3

In some cases the semantic requirements are presented as C++ code. Such code is intended as a specification of equivalence of a construct to another construct, not necessarily as the way the construct must be implemented.[161]

17.14.0.1 Detailed specifications [structure.specifications]

1 The detailed specifications each contain the following elements:

(1.1) — name and brief description

(1.2) — synopsis (class definition or function prototype, as appropriate)

(1.3) — restrictions on template arguments, if any

161) Although in some cases the code given is unambiguously the optimum implementation.

(1.4) — description of class invariants

(1.5) — description of function semantics

2 Descriptions of class member functions follow the order (as appropriate):[162]

(2.1) — constructor(s) and destructor

(2.2) — copying, moving & assignment functions

(2.3) — comparison functions

(2.4) — modifier functions

(2.5) — observer functions

(2.6) — operators and other non-member functions

3 Descriptions of function semantics contain the following elements (as appropriate):[163]

(3.1) — *Requires:* the preconditions for calling the function

(3.2) — *Effects:* the actions performed by the function

(3.3) — *Synchronization:* the synchronization operations (1.859) applicable to the function

(3.4) — *Postconditions:* the observable results established by the function

(3.5) — *Returns:* a description of the value(s) returned by the function

(3.6) — *Throws:* any exceptions thrown by the function, and the conditions that would cause the exception

(3.7) — *Complexity:* the time and/or space complexity of the function

(3.8) — *Remarks:* additional semantic constraints on the function

(3.9) — *Error conditions:* the error conditions for error codes reported by the function.

(3.10) — *Notes:* non-normative comments about the function

4 Whenever the *Effects:* element specifies that the semantics of some function `F` are *Equivalent to* some code sequence, then the various elements are interpreted as follows. If `F`'s semantics specifies a *Requires:* element, then that requirement is logically imposed prior to the *equivalent-to* semantics. Next, the semantics of the code sequence are determined by the *Requires:*, *Effects:*, *Postconditions:*, *Returns:*, *Throws:*, *Complexity:*, *Remarks:*, *Error conditions:*, and *Notes:* specified for the function invocations contained in the code sequence. The value returned from `F` is specified by `F`'s *Returns:* element, or if `F` has no *Returns:* element, a non-`void` return from `F` is specified by the *Returns:* elements in the code sequence. If `F`'s semantics contains a *Throws:*, *Postconditions:*, or *Complexity:* element, then that supersedes any occurrences of that element in the code sequence.

5 For non-reserved replacement and handler functions, Clause 18 specifies two behaviors for the functions in question: their required and default behavior. The *default behavior* describes a function definition provided by the implementation. The *required behavior* describes the semantics of a function definition provided by either the implementation or a C++ program. Where no distinction is explicitly made in the description, the behavior described is the required behavior.

162) To save space, items that do not apply to a class are omitted. For example, if a class does not specify any comparison functions, there will be no "Comparison functions" subclause.

163) To save space, items that do not apply to a function are omitted. For example, if a function does not specify any further preconditions, there will be no "Requires" paragraph.

6 If the formulation of a complexity requirement calls for a negative number of operations, the actual requirement is zero operations.[164]

7

Complexity requirements specified in the library clauses are upper bounds, and implementations that provide better complexity guarantees satisfy the requirements.

1 Error conditions specify conditions where a function may fail. The conditions are listed, together with a suitable explanation, as the `enum class errc` constants (19.14).

17.17.0.1 C library [structure.see.also]

1 Paragraphs labeled "See also:" contain cross-references to the relevant portions of this International Standard and the ISO C standard, which is incorporated into this International Standard by reference.

17.17.1 Other conventions [conventions]

1 This subclause describes several editorial conventions used to describe the contents of the C++ standard library. These conventions are for describing implementation-defined types (17.17.1.1), and member functions (17.28.0.1).

17.17.1.1 Type descriptions [type.descriptions]

17.17.1.1.1 General [type.descriptions.general]

1 The Requirements subclauses may describe names that are used to specify constraints on template arguments.[165] These names are used in library Clauses to describe the types that may be supplied as arguments by a C++ program when instantiating template components from the library.

2 Certain types defined in Clause 27 are used to describe implementation-defined types. They are based on other types, but with added constraints.

17.17.1.1.2 Enumerated types [enumerated.types]

1

Several types defined in Clause 27 are *enumerated types*. Each enumerated type may be implemented as an enumeration or as a synonym for an enumeration.[166]

1

The enumerated type *enumerated* can be written:

```
enum enumerated { V0, V1, V2, V3, ..... };

static const enumerated C0 (V0);
static const enumerated C1 (V1);
static const enumerated C2 (V2);
static const enumerated C3 (V3);
  .....
```

1

Here, the names *C0*, *C1*, etc. represent *enumerated elements* for this particular enumerated type. All such elements have distinct values.

164) This simplifies the presentation of complexity requirements in some cases.
165) Examples from 17.128.1 include: `EqualityComparable`, `LessThanComparable`, `CopyConstructible`. Examples from 24.2 include: `InputIterator`, `ForwardIterator`, `Function`, `Predicate`.
166) Such as an integer type, with constant integer values (3.3151.1).

17.20.0.0.1 Bitmask types [bitmask.types]

1

Several types defined in Clauses 18 through 30 and Annex D are *bitmask types*. Each bitmask type can be implemented as an enumerated type that overloads certain operators, as an integer type, or as a `bitset` (20.68).

1 The bitmask type *bitmask* can be written:

```
// For exposition only.
// int_type is an integral type capable of
// representing all values of the bitmask type.
enum bitmask : int_type {
  V0 = 1 << 0, V1 = 1 << 1, V2 = 1 << 2, V3 = 1 << 3, .....
};

constexpr bitmask C0(V0);
constexpr bitmask C1(V1);
constexpr bitmask C2(V2);
constexpr bitmask C3(V3);
  .....

constexpr bitmask operator&(bitmask X, bitmask Y) {
  return static_cast<bitmask>(
    static_cast<int_type>(X) & static_cast<int_type>(Y));
}
constexpr bitmask operator|(bitmask X, bitmask Y) {
  return static_cast<bitmask>(
    static_cast<int_type>(X) | static_cast<int_type>(Y));
}
constexpr bitmask operator^(bitmask X, bitmask Y){
  return static_cast<bitmask>(
    static_cast<int_type>(X) ^ static_cast<int_type>(Y));
}
constexpr bitmask operator~(bitmask X){
  return static_cast<bitmask>(~static_cast<int_type>(X));
}
bitmask& operator&=(bitmask& X, bitmask Y){
  X = X & Y; return X;
}
bitmask& operator|=(bitmask& X, bitmask Y) {
  X = X | Y; return X;
}
bitmask& operator^=(bitmask& X, bitmask Y) {
  X = X ^ Y; return X;
}
```

2 Here, the names *C0*, *C1*, etc. represent *bitmask elements* for this particular bitmask type. All such elements have distinct, nonzero values such that, for any pair *Ci* and *Cj* where $i \mathrel{!=} j$, *Ci* & *Ci* is nonzero and *Ci* & *Cj* is zero. Additionally, the value 0 is used to represent an *empty bitmask*, in which no bitmask elements are set.

3

The following terms apply to objects and values of bitmask types:

— To *set* a value *Y* in an object *X* is to evaluate the expression *X* `|=` *Y*.

— To *clear* a value Y in an object X is to evaluate the expression X `&= ~` Y.

— The value Y *is set* in the object X if the expression X `&` Y is nonzero.

17.24.0.0.1 Character sequences [character.seq]

1

The C standard library makes widespread use of characters and character sequences that follow a few uniform conventions:

— A *letter* is any of the 26 lowercase or 26 uppercase letters in the basic execution character set.[167]

— The *decimal-point character* is the (single-byte) character used by functions that convert between a (single-byte) character sequence and a value of one of the floating-point types. It is used in the character sequence to denote the beginning of a fractional part. It is represented in Clauses 18 through 30 and Annex D by a period, `'.'`, which is also its value in the `"C"` locale, but may change during program execution by a call to `setlocale(int, const char*)`,[168] or by a change to a `locale` object, as described in Clauses 22.3 and 27.

— A *character sequence* is an array object (8.251.1) A that can be declared as `T A[N]`, where T is any of the types `char`, `unsigned char`, or `signed char` (3.3151.1), optionally qualified by any combination of `const` or `volatile`. The initial elements of the array have defined contents up to and including an element determined by some predicate. A character sequence can be designated by a pointer value S that points to its first element.

17.25.0.0.0.1 Byte strings [byte.strings]

1 A *null-terminated byte string*, or NTBS, is a character sequence whose highest-addressed element with defined content has the value zero (the *terminating null* character); no other element in the sequence has the value zero.[169]

2

The *length* of an NTBS is the number of elements that precede the terminating null character. An *empty* NTBS has a length of zero.

1 The *value* of an NTBS is the sequence of values of the elements up to and including the terminating null character.

2 A *static* NTBS is an NTBS with static storage duration.[170]

17.26.0.0.0.1 Multibyte strings [multibyte.strings]

1

A *null-terminated multibyte string,* or NTMBS, is an NTBS that constitutes a sequence of valid multibyte characters, beginning and ending in the initial shift state.[171]

1 A *static* NTMBS is an NTMBS with static storage duration.

167) Note that this definition differs from the definition in ISO C 7.1.1.
168) declared in `<clocale>` (22.118).
169) Many of the objects manipulated by function signatures declared in `<cstring>` (21.261) are character sequences or NTBSs. The size of some of these character sequences is limited by a length value, maintained separately from the character sequence.
170) A string literal, such as `"abc"`, is a static NTBS.
171) An NTBS that contains characters only from the basic execution character set is also an NTMBS. Each multibyte character then consists of a single byte.

17.28.0.1 Functions within classes [functions.within.classes]

1 For the sake of exposition, Clauses 18 through 30 and Annex D do not describe copy/move constructors, assignment operators, or (non-virtual) destructors with the same apparent semantics as those that can be generated by default (12.30, 12.807, 12.1577).

2 It is unspecified whether the implementation provides explicit definitions for such member function signatures, or for virtual destructors that can be generated by default.

17.28.0.2 Private members [objects.within.classes]

1

Clauses 18 through 30 and Annex D do not specify the representation of classes, and intentionally omit specification of class members (9.194). An implementation may define static or non-static class members, or both, as needed to implement the semantics of the member functions specified in Clauses 18 through 30 and Annex D.

1

Objects of certain classes are sometimes required by the external specifications of their classes to store data, apparently in member objects. For the sake of exposition, some subclauses provide representative declarations, and semantic requirements, for private member objects of classes that meet the external specifications of the classes. The declarations for such member objects and the definitions of related member types are followed by a comment that ends with *exposition only*, as in:

```
streambuf* sb;   // exposition only
```

1

An implementation may use any technique that provides equivalent external behavior.

17.37 Library-wide requirements [requirements]

1 This subclause specifies requirements that apply to the entire C++ standard library. Clauses 18 through 30 and Annex D specify the requirements of individual entities within the library.

2 Requirements specified in terms of interactions between threads do not apply to programs having only a single thread of execution.

3 Within this subclause, 17.37.1 describes the library's contents and organization, 17.109.1 describes how well-formed C++ programs gain access to library entities, 17.128.1 describes constraints on types and functions used with the C++ standard library, 17.194.1 describes constraints on well-formed C++ programs, and 17.544.1 describes constraints on conforming implementations.

17.37.1 Library contents and organization [organization]

1 17.37.1.1 describes the entities defined in the C++ standard library. 17.48.0.1 lists the standard library headers and some constraints on those headers. 17.97.0.1 lists requirements for a freestanding implementation of the C++ standard library.

17.37.1.1 Library contents [contents]

1

The C++ standard library provides definitions for the following types of entities: macros, values, types, templates, classes, functions, objects.

1

All library entities except macros, `operator new` and `operator delete` are defined within the namespace `std` or namespaces nested within namespace `std`.[172] It is unspecified whether names declared in a specific namespace are declared directly in that namespace or in an inline namespace inside that namespace.[173]

1 Whenever a name `x` defined in the standard library is mentioned, the name `x` is assumed to be fully qualified as `::std::x`, unless explicitly described otherwise. For example, if the Effects section for library function `F` is described as calling library function `G`, the function `::std::G` is meant.

17.48.0.1 Headers [headers]

1

Each element of the C++ standard library is declared or defined (as appropriate) in a *header*.[174]

1

The C++ standard library provides 53 *C++ library headers*, as shown in Table 14.

Table 14 — C++ library headers

`<algorithm>`	`<fstream>`	`<list>`	`<regex>`	`<tuple>`
`<array>`	`<functional>`	`<locale>`	`<scoped_allocator>`	`<type_traits>`
`<atomic>`	`<future>`	`<map>`	`<set>`	`<typeindex>`
`<bitset>`	`<initializer_list>`	`<memory>`	`<sstream>`	`<typeinfo>`
`<chrono>`	`<iomanip>`	`<mutex>`	`<stack>`	`<unordered_map>`
`<codecvt>`	`<ios>`	`<new>`	`<stdexcept>`	`<unordered_set>`
`<complex>`	`<iosfwd>`	`<numeric>`	`<streambuf>`	`<utility>`
`<condition_variable>`	`<iostream>`	`<ostream>`	`<string>`	`<valarray>`
`<deque>`	`<istream>`	`<queue>`	`<strstream>`	`<vector>`
`<exception>`	`<iterator>`	`<random>`	`<system_error>`	
`<forward_list>`	`<limits>`	`<ratio>`	`<thread>`	

1

The facilities of the C standard Library are provided in 26 additional headers, as shown in Table 15.

Table 15 — C++ headers for C library facilities

`<cassert>`	`<cinttypes>`	`<csignal>`	`<cstdio>`	`<cwchar>`
`<ccomplex>`	`<ciso646>`	`<cstdalign>`	`<cstdlib>`	`<cwctype>`
`<cctype>`	`<climits>`	`<cstdarg>`	`<cstring>`	
`<cerrno>`	`<clocale>`	`<cstdbool>`	`<ctgmath>`	
`<cfenv>`	`<cmath>`	`<cstddef>`	`<ctime>`	
`<cfloat>`	`<csetjmp>`	`<cstdint>`	`<cuchar>`	

1

Except as noted in Clauses 18 through 30 and Annex D, the contents of each header `c`*`name`* shall be the same as that of the corresponding header *`name`*`.h`, as specified in the C standard library (1.4) or the C Unicode TR, as appropriate, as if by inclusion. In the C++ standard library, however, the declarations (except for

172) The C standard library headers (Annex D.9) also define names within the global namespace, while the C++ headers for C library facilities (17.48.0.1) may also define names within the global namespace.
173) This gives implementers freedom to use inline namespaces to support multiple configurations of the library.
174) A header is not necessarily a source file, nor are the sequences delimited by `<` and `>` in header names necessarily valid source file names (16.48).

names which are defined as macros in C) are within namespace scope (3.736.1) of the namespace `std`. It is unspecified whether these names are first declared within the global namespace scope and are then injected into namespace `std` by explicit *using-declarations* (7.1311.1).

1

Names which are defined as macros in C shall be defined as macros in the C++ standard library, even if C grants license for implementation as functions. [*Note:* The names defined as macros in C include the following: `assert`, `offsetof`, `setjmp`, `va_arg`, `va_end`, and `va_start`. — *end note*]

1

Names that are defined as functions in C shall be defined as functions in the C++ standard library.[175]

1

Identifiers that are keywords or operators in C++ shall not be defined as macros in C++ standard library headers.[176]

1 D.9, C standard library headers, describes the effects of using the *`name`*`.h` (C header) form in a C++ program.[177]

17.97.0.1 Freestanding implementations [compliance]

1

Two kinds of implementations are defined: *hosted* and *freestanding* (1.196). For a hosted implementation, this International Standard describes the set of available headers.

1

A freestanding implementation has an implementation-defined set of headers. This set shall include at least the headers shown in Table 16.

Table 16 — C++ headers for freestanding implementations

Subclause		Header(s)
		`<ciso646>`
18.2	Types	`<cstddef>`
18.41	Implementation properties	`<cfloat>` `<limits>` `<climits>`
18.114	Integer types	`<cstdint>`
18.127	Start and termination	`<cstdlib>`
18.159	Dynamic memory management	`<new>`
18.227	Type identification	`<typeinfo>`
18.241	Exception handling	`<exception>`
18.274	Initializer lists	`<initializer_list>`
18.295	Other runtime support	`<cstdalign>` `<cstdarg>` `<cstdbool>`
20.449	Type traits	`<type_traits>`
29	Atomics	`<atomic>`

1

175) This disallows the practice, allowed in C, of providing a masking macro in addition to the function prototype. The only way to achieve equivalent inline behavior in C++ is to provide a definition as an extern inline function.
176) In particular, including the standard header `<iso646.h>` or `<ciso646>` has no effect.
177) The `".h"` headers dump all their names into the global namespace, whereas the newer forms keep their names in namespace `std`. Therefore, the newer forms are the preferred forms for all uses except for C++ programs which are intended to be strictly compatible with C.

The supplied version of the header `<cstdlib>` shall declare at least the functions `abort`, `atexit`, `at_quick_exit`, `exit`, and `quick_exit` (18.127). The other headers listed in this table shall meet the same requirements as for a hosted implementation.

17.109.1 Using the library [using]

17.109.1.1 Overview [using.overview]

1

This section describes how a C++ program gains access to the facilities of the C++ standard library. 17.110.0.1 describes effects during translation phase 4, while 17.125.0.1 describes effects during phase 8 (2.6).

17.110.0.1 Headers [using.headers]

1 The entities in the C++ standard library are defined in headers, whose contents are made available to a translation unit when it contains the appropriate `#include` preprocessing directive (16.48).

2

A translation unit may include library headers in any order (Clause 2). Each may be included more than once, with no effect different from being included exactly once, except that the effect of including either `<cassert>` or `<assert.h>` depends each time on the lexically current definition of `NDEBUG`.[178]

1

A translation unit shall include a header only outside of any external declaration or definition, and shall include the header lexically before the first reference in that translation unit to any of the entities declared in that header. No diagnostic is required.

17.125.0.1 Linkage [using.linkage]

1

Entities in the C++ standard library have external linkage (3.1459). Unless otherwise specified, objects and functions have the default `extern "C++"` linkage (7.1477).

1

Whether a name from the C standard library declared with external linkage has `extern "C"` or `extern "C++"` linkage is implementation-defined. It is recommended that an implementation use `extern "C++"` linkage for this purpose.[179]

1 Objects and functions defined in the library and required by a C++ program are included in the program prior to program startup.

See also: replacement functions (17.428.0.3), run-time changes (17.436.0.1).

17.128.1 Requirements on types and expressions [utility.requirements]

1 17.128.1.1 describes requirements on types and expressions used to instantiate templates defined in the C++ standard library. 17.143.0.1 describes the requirements on swappable types and swappable expressions. 17.153.0.1 describes the requirements on pointer-like types that support null values. 17.159.0.1 describes the requirements on hash function objects. 17.165.0.1 describes the requirements on storage allocators.

17.128.1.1 Template argument requirements [utility.arg.requirements]

1

178) This is the same as the Standard C library.

179) The only reliable way to declare an object or function signature from the Standard C library is by including the header that declares it, notwithstanding the latitude granted in 7.1.4 of the C Standard.

The template definitions in the C++ standard library refer to various named requirements whose details are set out in tables 17–24. In these tables, `T` is an object or reference type to be supplied by a C++ program instantiating a template; `a`, `b`, and `c` are values of type (possibly `const`) `T`; `s` and `t` are modifiable lvalues of type `T`; `u` denotes an identifier; `rv` is an rvalue of type `T`; and `v` is an lvalue of type (possibly `const`) `T` or an rvalue of type `const T`.

1

In general, a default constructor is not required. Certain container class member function signatures specify `T()` as a default argument. `T()` shall be a well-defined expression (8.821) if one of those signatures is called using the default argument (8.545.1).

Table 17 — `EqualityComparable` requirements [**equalitycomparable**]

Expression	Return type	Requirement
`a == b`	convertible to `bool`	`==` is an equivalence relation, that is, it has the following properties: — For all `a`, `a == a`. — If `a == b`, then `b == a`. — If `a == b` and `b == c`, then `a == c`.

Table 18 — `LessThanComparable` requirements [**lessthancomparable**]

Expression	Return type	Requirement
`a < b`	convertible to `bool`	`<` is a strict weak ordering relation (25.98)

Table 19 — `DefaultConstructible` requirements [**defaultconstructible**]

Expression	Post-condition
`T t;`	object `t` is default-initialized
`T u{};`	object `u` is value-initialized
`T()` `T{}`	a temporary object of type `T` is value-initialized

Table 20 — `MoveConstructible` requirements [**moveconstructible**]

Expression	Post-condition
`T u = rv;`	`u` is equivalent to the value of `rv` before the construction
`T(rv)`	`T(rv)` is equivalent to the value of `rv` before the construction
`rv`'s state is unspecified [*Note:* `rv` must still meet the requirements of the library component that is using it. The operations listed in those requirements must work as specified whether `rv` has been moved from or not. — *end note*]	

Table 21 — `CopyConstructible` requirements (in addition to `MoveConstructible`) [**copyconstructible**]

Expression	Post-condition
`T u = v;`	the value of `v` is unchanged and is equivalent to `u`
`T(v)`	the value of `v` is unchanged and is equivalent to `T(v)`

Table 22 — `MoveAssignable` requirements [**moveassignable**]

Expression	Return type	Return value	Post-condition
`t = rv`	`T&`	`t`	`t` is equivalent to the value of `rv` before the assignment
`rv`'s state is unspecified. [*Note:* `rv` must still meet the requirements of the library component that is using it. The operations listed in those requirements must work as specified whether `rv` has been moved from or not. — *end note*]			

Table 23 — `CopyAssignable` requirements (in addition to `MoveAssignable`) [**copyassignable**]

Expression	Return type	Return value	Post-condition
`t = v`	`T&`	`t`	`t` is equivalent to `v`, the value of `v` is unchanged

Table 24 — `Destructible` requirements [**destructible**]

Expression	Post-condition
`u.~T()`	All resources owned by `u` are reclaimed, no exception is propagated.

17.143.0.1 `Swappable` requirements [**swappable.requirements**]

1 This subclause provides definitions for swappable types and expressions. In these definitions, let `t` denote an expression of type `T`, and let `u` denote an expression of type `U`.

2 An object `t` is *swappable with* an object `u` if and only if:

(2.1) — the expressions `swap(t, u)` and `swap(u, t)` are valid when evaluated in the context described below, and

(2.2) — these expressions have the following effects:

(2.2.1) — the object referred to by `t` has the value originally held by `u` and

(2.2.2) — the object referred to by `u` has the value originally held by `t`.

3

The context in which `swap(t, u)` and `swap(u, t)` are evaluated shall ensure that a binary non-member function named "swap" is selected via overload resolution (13.88) on a candidate set that includes:

— the two `swap` function templates defined in `<utility>` (20.2) and

— the lookup set produced by argument-dependent lookup (3.1092.1).

[*Note:* If `T` and `U` are both fundamental types or arrays of fundamental types and the declarations from the header `<utility>` are in scope, the overall lookup set described above is equivalent to that of the qualified name lookup applied to the expression `std::swap(t, u)` or `std::swap(u, t)` as appropriate. — *end note*]

[*Note:* It is unspecified whether a library component that has a swappable requirement includes the header `<utility>` to ensure an appropriate evaluation context. — *end note*]

1

An rvalue or lvalue `t` is *swappable* if and only if `t` is swappable with any rvalue or lvalue, respectively, of type `T`.

1 A type `X` satisfying any of the iterator requirements (24.2) satisfies the requirements of `ValueSwappable` if, for any dereferenceable object `x` of type `X`, `*x` is swappable.

[*Example:* User code can ensure that the evaluation of `swap` calls is performed in an appropriate context under the various conditions as follows:

```
#include <utility>

// Requires: std::forward<T>(t) shall be swappable with std::forward<U>(u).
template <class T, class U>
void value_swap(T&& t, U&& u) {
  using std::swap;
  swap(std::forward<T>(t), std::forward<U>(u)); // OK: uses "swappable with" conditions
                                                // for rvalues and lvalues
}

// Requires: lvalues of T shall be swappable.
template <class T>
void lv_swap(T& t1 T& t2) {
  using std::swap;
  swap(t1, t2);                                 // OK: uses swappable conditions for
}                                               // lvalues of type T

namespace N {
  struct A { int m; };
  struct Proxy { A* a; };
  Proxy proxy(A& a) { return Proxy{ &a }; }

  void swap(A& x, Proxy p) {
    std::swap(x.m, p.a->m);                     // OK: uses context equivalent to swappable
                                                // conditions for fundamental types
  }
  void swap(Proxy p, A& x) { swap(x, p); }      // satisfy symmetry constraint
}

int main() {
  int i = 1, j = 2;
  lv_swap(i, j);
  assert(i == 2 && j == 1);

  N::A a1 = { 5 }, a2 = { -5 };
  value_swap(a1, proxy(a2));
  assert(a1.m == -5 && a2.m == 5);
}
```

— *end example*]

17.153.0.1 `NullablePointer` requirements [nullablepointer.requirements]

1

A `NullablePointer` type is a pointer-like type that supports null values. A type `P` meets the requirements of `NullablePointer` if:

— `P` satisfies the requirements of `EqualityComparable`, `DefaultConstructible`, `CopyConstructible`, `CopyAssignable`, and `Destructible`,

— lvalues of type `P` are swappable (17.143.0.1),

— the expressions shown in Table 25 are valid and have the indicated semantics, and

— `P` satisfies all the other requirements of this subclause.

1

A value-initialized object of type `P` produces the null value of the type. The null value shall be equivalent only to itself. A default-initialized object of type `P` may have an indeterminate value. [*Note:* Operations involving indeterminate values may cause undefined behavior. — *end note*]

1

An object `p` of type `P` can be contextually converted to `bool` (Clause 4). The effect shall be as if `p != nullptr` had been evaluated in place of `p`.

1 No operation which is part of the `NullablePointer` requirements shall exit via an exception.

2

In Table 25, `u` denotes an identifier, `t` denotes a non-`const` lvalue of type `P`, `a` and `b` denote values of type (possibly `const`) `P`, and `np` denotes a value of type (possibly `const`) `std::nullptr_t`.

Table 25 — `NullablePointer` requirements **[nullablepointer]**

Expression	Return type	Operational semantics
`P u(np);` `P u = np;`		post: `u == nullptr`
`P(np)`		post: `P(np) == nullptr`
`t = np`	`P&`	post: `t == nullptr`
`a != b`	contextually convertible to `bool`	`!(a == b)`
`a == np` `np == a`	contextually convertible to `bool`	`a == P()`
`a != np` `np != a`	contextually convertible to `bool`	`!(a == np)`

17.159.0.1 Hash requirements [hash.requirements]

1 A type `H` meets the `Hash` requirements if:

(1.1) — it is a function object type (20.283),

(1.2) — it satisfies the requirements of `CopyConstructible` and `Destructible` (17.128.1.1), and

(1.3) — the expressions shown in Table 26 are valid and have the indicated semantics.

2

Given `Key` is an argument type for function objects of type `H`, in Table 26 `h` is a value of type (possibly `const`) `H`, `u` is an lvalue of type `Key`, and `k` is a value of a type convertible to (possibly `const`) `Key`.

Table 26 — `Hash` requirements **[hash]**

Expression	Return type	Requirement
`h(k)`	`size_t`	The value returned shall depend only on the argument `k` for the duration of the program. [*Note:* Thus all evaluations of the expression `h(k)` with the same value for `k` yield the same result for a given execution of the program. — *end note*] [*Note:* For two different values `t1` and `t2`, the probability that `h(t1)` and `h(t2)` compare equal should be very small, approaching `1.0 / numeric_limits<size_t>::max()`. — *end note*]
`h(u)`	`size_t`	Shall not modify `u`.

17.165.0.1 Allocator requirements [allocator.requirements]

1 The library describes a standard set of requirements for *allocators*, which are class-type objects that encapsulate the information about an allocation model. This information includes the knowledge of pointer types, the type of their difference, the type of the size of objects in this allocation model, as well as the memory allocation and deallocation primitives for it. All of the string types (Clause 21), containers (Clause 23) (except array), string buffers and string streams (Clause 27), and `match_results` (Clause 28) are parameterized in terms of allocators.

2

The template struct `allocator_traits` (20.86.3) supplies a uniform interface to all allocator types. Table 27 describes the types manipulated through allocators. Table 28 describes the requirements on allocator types and thus on types used to instantiate `allocator_traits`. A requirement is optional if the last column of Table 28 specifies a default for a given expression. Within the standard library `allocator_traits` template, an optional requirement that is not supplied by an allocator is replaced by the specified default expression. A user specialization of `allocator_traits` may provide different defaults and may provide defaults for different requirements than the primary template. Within Tables 27 and 28, the use of `move` and `forward` always refers to `std::move` and `std::forward`, respectively.

Table 27 — Descriptive variable definitions

Variable	Definition
`T, U, C`	any non-const object type (3.2955)
`V`	a type convertible to `T`
`X`	an Allocator class for type `T`
`Y`	the corresponding Allocator class for type `U`
`XX`	the type `allocator_traits<X>`
`YY`	the type `allocator_traits<Y>`
`t`	a value of type `const T&`
`a, a1, a2`	values of type `X&`
`a3`	an rvalue of type `X`
`b`	a value of type `Y`
`c`	a pointer of type `C*` through which indirection is valid
`p`	a value of type `XX::pointer`, obtained by calling `a1.allocate`, where `a1 == a`
`q`	a value of type `XX::const_pointer` obtained by conversion from a value `p`.

Table 27 — Descriptive variable definitions (continued)

Variable	Definition
w	a value of type XX::void_pointer obtained by conversion from a value p
z	a value of type XX::const_void_pointer obtained by conversion from a value q or a value w
r	a value of type T& obtained by the expression *p.
s	a value of type const T& obtained by the expression *q or by conversion from a value r.
u	a value of type XX:const_void_pointer obtained by conversion from a result value of YY::allocate, or else a value of type (possibly const) std::nullptr_t.
v	a value of type V
n	a value of type XX::size_type.
Args	a template parameter pack
args	a function parameter pack with the pattern Args&&

Table 28 — Allocator requirements

Expression	Return type	Assertion/note pre-/post-condition	Default
X::pointer			T*
X::const_pointer		X::pointer is convertible to X::const_pointer	pointer_traits<X::pointer>::rebind<const T>
X::void_pointer Y::void_pointer		X::pointer is convertible to X::void_pointer. X::void_pointer and Y::void_pointer are the same type.	pointer_traits<X::pointer>::rebind<void>
X::const_void_pointer Y::const_void_pointer		X::pointer, X::const_pointer, and X::void_pointer are convertible to X::const_void_pointer. X::const_void_pointer and Y::const_void_pointer are the same type.	pointer_traits<X::pointer>::rebind<const void>
X::value_type	Identical to T		
X::size_type	unsigned integer type	a type that can represent the size of the largest object in the allocation model.	make_unsigned_t<X::difference_type>

Table 28 — Allocator requirements (continued)

Expression	Return type	Assertion/note pre-/post-condition	Default
`X::difference_type`	signed integer type	a type that can represent the difference between any two pointers in the allocation model.	`pointer_traits<X::pointer>::difference_type`
`typename X::template rebind<U>::other`	`Y`	For all `U` (including `T`), `Y::template rebind<T>::other` is `X`.	See Note A, below.
`*p`	`T&`		
`*q`	`const T&`	`*q` refers to the same object as `*p`	
`p->m`	type of `T::m`	*pre:* `(*p).m` is well-defined. equivalent to `(*p).m`	
`q->m`	type of `T::m`	*pre:* `(*q).m` is well-defined. equivalent to `(*q).m`	
`static_cast<X::pointer>(w)`	`X::pointer`	`static_cast<X::pointer>(w) == p`	
`static_cast<X::const_pointer>(z)`	`X::const_pointer`	`static_cast<X::const_pointer>(z) == q`	
`a.allocate(n)`	`X::pointer`	Memory is allocated for `n` objects of type `T` but objects are not constructed. `allocate` may raise an appropriate exception.[180] [*Note:* If `n == 0`, the return value is unspecified. — *end note*]	
`a.allocate(n, u)`	`X::pointer`	Same as `a.allocate(n)`. The use of `u` is unspecified, but it is intended as an aid to locality.	`a.allocate(n)`
`a.deallocate(p,n)`	(not used)	All `n T` objects in the area pointed to by `p` shall be destroyed prior to this call. `n` shall match the value passed to `allocate` to obtain this memory. Does not throw exceptions. [*Note:* `p` shall not be singular. — *end note*]	
`a.max_size()`	`X::size_type`	the largest value that can meaningfully be passed to `X::allocate()`	`numeric_limits<size_type>::max()`
`a1 == a2`	`bool`	returns `true` only if storage allocated from each can be deallocated via the other. `operator==` shall be reflexive, symmetric, and transitive, and shall not exit via an exception.	

Table 28 — Allocator requirements (continued)

Expression	Return type	Assertion/note pre-/post-condition	Default
`a1 != a2`	`bool`	same as `!(a1 == a2)`	
`a == b`	`bool`	same as `a == Y::rebind<T>::other(b)`	
`a != b`	`bool`	same as `!(a == b)`	
`X a1(a);` `X a1 = a;`		Shall not exit via an exception. post: `a1 == a`	
`X a(b);`		Shall not exit via an exception. post: `Y(a) == b`, `a == X(b)`	
`X a1(move(a));` `X a1 = move(a);`		Shall not exit via an exception. post: `a1` equals the prior value of `a`.	
`X a(move(b));`		Shall not exit via an exception. post: `a` equals the prior value of `X(b)`.	
`a.construct(c, args)`	(not used)	Effect: Constructs an object of type `C` at `c`	`::new ((void*)c) C(forward< Args> (args)...)`
`a.destroy(c)`	(not used)	Effect: Destroys the object at `c`	`c->~C()`
`a.select_on_container_copy_construction()`	`X`	Typically returns either `a` or `X()`	`return a;`
`X::propagate_on_container_copy_assignment`	Identical to or derived from `true_type` or `false_type`	`true_type` only if an allocator of type `X` should be copied when the client container is copy-assigned.	`false_type`
`X::propagate_on_container_move_assignment`	Identical to or derived from `true_type` or `false_type`	`true_type` only if an allocator of type `X` should be moved when the client container is move-assigned.	`false_type`
`X::propagate_on_container_swap`	Identical to or derived from `true_type` or `false_type`	`true_type` only if an allocator of type `X` should be swapped when the client container is swapped.	`false_type`

1

Note A: The member class template `rebind` in the table above is effectively a typedef template. [*Note:* In general, if the name `Allocator` is bound to `SomeAllocator<T>`, then `Allocator::rebind<U>::other` is the same type as `SomeAllocator<U>`, where `SomeAllocator<T>::value_type` is `T` and `SomeAllocator<U>::value_type` is `U`. — *end note*] If `Allocator` is a class template instantiation of the form `SomeAllocator<T, Args>`, where `Args` is zero or more type arguments, and `Allocator` does not supply a `rebind` member template, the standard `allocator_traits` template uses `SomeAllocator<U, Args>` in place of `Allocator::`

180) It is intended that `a.allocate` be an efficient means of allocating a single object of type `T`, even when `sizeof(T)` is small. That is, there is no need for a container to maintain its own free list.

`rebind<U>::other` by default. For allocator types that are not template instantiations of the above form, no default is provided.

1

An allocator type `X` shall satisfy the requirements of `CopyConstructible` (17.128.1.1). The `X::pointer`, `X::const_pointer`, `X::void_pointer`, and `X::const_void_pointer` types shall satisfy the requirements of `NullablePointer` (17.153.0.1). No constructor, comparison operator, copy operation, move operation, or swap operation on these types shall exit via an exception. `X::pointer` and `X::const_pointer` shall also satisfy the requirements for a random access iterator (24.2).

1 Let `x1` and `x2` denote objects of (possibly different) types `X::void_pointer`, `X::const_void_pointer`, `X::pointer`, or `X::const_pointer`. Then, `x1` and `x2` are *equivalently-valued* pointer values, if and only if both `x1` and `x2` can be explicitly converted to the two corresponding objects `px1` and `px2` of type `X::const_pointer`, using a sequence of `static_cast`s using only these four types, and the expression `px1 == px2` evaluates to `true`.

2 Let `w1` and `w2` denote objects of type `X::void_pointer`. Then for the expressions

```
w1 == w2
w1 != w2
```

either or both objects may be replaced by an equivalently-valued object of type `X::const_void_pointer` with no change in semantics.

3 Let `p1` and `p2` denote objects of type `X::pointer`. Then for the expressions

```
p1 == p2
p1 != p2
p1 < p2
p1 <= p2
p1 >= p2
p1 > p2
p1 - p2
```

either or both objects may be replaced by an equivalently-valued object of type `X::const_pointer` with no change in semantics.

4

An allocator may constrain the types on which it can be instantiated and the arguments for which its `construct` member may be called. If a type cannot be used with a particular allocator, the allocator class or the call to `construct` may fail to instantiate.

[*Example:* the following is an allocator class template supporting the minimal interface that satisfies the requirements of Table 28:

```
template <class Tp>
struct SimpleAllocator {
  typedef Tp value_type;
  SimpleAllocator(ctor args);

  template <class T> SimpleAllocator(const SimpleAllocator<T>& other);

  Tp* allocate(std::size_t n);
  void deallocate(Tp* p, std::size_t n);
};

template <class T, class U>
bool operator==(const SimpleAllocator<T>&, const SimpleAllocator<U>&);
```

```
template <class T, class U>
bool operator!=(const SimpleAllocator<T>&, const SimpleAllocator<U>&);
```

— *end example*]

1

If the alignment associated with a specific over-aligned type is not supported by an allocator, instantiation of the allocator for that type may fail. The allocator also may silently ignore the requested alignment. [*Note:* Additionally, the member function `allocate` for that type may fail by throwing an object of type `std::bad_alloc`. — *end note*]

17.194.1 Constraints on programs [constraints]

17.194.1.1 Overview [constraints.overview]

1

This section describes restrictions on C++ programs that use the facilities of the C++ standard library. The following subclauses specify constraints on the program's use of namespaces (17.198.0.1.1), its use of various reserved names (17.296.0.1), its use of headers (17.428.0.1), its use of standard library classes as base classes (17.428.0.2), its definitions of replacement functions (17.428.0.3), and its installation of handler functions during execution (17.436.0.1).

17.198.0.1 Namespace use [namespace.constraints]

17.198.0.1.1 Namespace `std` [namespace.std]

1

The behavior of a C++ program is undefined if it adds declarations or definitions to namespace `std` or to a namespace within namespace `std` unless otherwise specified. A program may add a template specialization for any standard library template to namespace `std` only if the declaration depends on a user-defined type and the specialization meets the standard library requirements for the original template and is not explicitly prohibited.[181]

1

The behavior of a C++ program is undefined if it declares

— an explicit specialization of any member function of a standard library class template, or

— an explicit specialization of any member function template of a standard library class or class template, or

— an explicit or partial specialization of any member class template of a standard library class or class template.

A program may explicitly instantiate a template defined in the standard library only if the declaration depends on the name of a user-defined type and the instantiation meets the standard library requirements for the original template.

1 A translation unit shall not declare namespace `std` to be an inline namespace (7.1149.1).

17.295.0.0.1 Namespace `posix` [namespace.posix]

1

181) Any library code that instantiates other library templates must be prepared to work adequately with any user-supplied specialization that meets the minimum requirements of the Standard.

The behavior of a C++ program is undefined if it adds declarations or definitions to namespace `posix` or to a namespace within namespace `posix` unless otherwise specified. The namespace `posix` is reserved for use by ISO/IEC 9945 and other POSIX standards.

17.296.0.1 Reserved names [reserved.names]

1

The C++ standard library reserves the following kinds of names:

— macros

— global names

— names with external linkage

1

If a program declares or defines a name in a context where it is reserved, other than as explicitly allowed by this Clause, its behavior is undefined.

17.309.0.0.1 Macro names [macro.names]

1

A translation unit that includes a standard library header shall not `#define` or `#undef` names declared in any standard library header.

1

A translation unit shall not `#define` or `#undef` names lexically identical to keywords, to the identifiers listed in Table 3, or to the *attribute-tokens* described in 7.1546.

17.332.0.0.1 Global names [global.names]

1

Certain sets of names and function signatures are always reserved to the implementation:

— Each name that contains a double underscore `__` or begins with an underscore followed by an uppercase letter (2.287) is reserved to the implementation for any use.

— Each name that begins with an underscore is reserved to the implementation for use as a name in the global namespace.

17.405.0.0.1 External linkage [extern.names]

1

Each name declared as an object with external linkage in a header is reserved to the implementation to designate that library object with external linkage,[182] both in namespace `std` and in the global namespace.

1 Each global function signature declared with external linkage in a header is reserved to the implementation to designate that function signature with external linkage. [183]

2

Each name from the Standard C library declared with external linkage is reserved to the implementation for use as a name with `extern "C"` linkage, both in namespace std and in the global namespace.

182) The list of such reserved names includes `errno`, declared or defined in `<cerrno>`.

183) The list of such reserved function signatures with external linkage includes `setjmp(jmp_buf)`, declared or defined in `<csetjmp>`, and `va_end(va_list)`, declared or defined in `<cstdarg>`.

1

Each function signature from the Standard C library declared with external linkage is reserved to the implementation for use as a function signature with both `extern "C"` and `extern "C++"` linkage, [184] or as a name of namespace scope in the global namespace.

17.420.0.0.1 Types [extern.types]

1

For each type T from the Standard C library,[185] the types `::T` and `std::T` are reserved to the implementation and, when defined, `::T` shall be identical to `std::T`.

17.424.0.0.1 User-defined literal suffixes [usrlit.suffix]

1

Literal suffix identifiers that do not start with an underscore are reserved for future standardization.

17.428.0.1 Headers [alt.headers]

1 If a file with a name equivalent to the derived file name for one of the C++ standard library headers is not provided as part of the implementation, and a file with that name is placed in any of the standard places for a source file to be included (16.48), the behavior is undefined.

17.428.0.2 Derived classes [derived.classes]

1 Virtual member function signatures defined for a base class in the C++ standard library may be overridden in a derived class defined in the program (10.113).

17.428.0.3 Replacement functions [replacement.functions]

1

Clauses 18 through 30 and Annex D describe the behavior of numerous functions defined by the C++ standard library. Under some circumstances, however, certain of these function descriptions also apply to replacement functions defined in the program (17.9).

1

A C++ program may provide the definition for any of twelve dynamic memory allocation function signatures declared in header `<new>` (3.2331.1, 18.159):

— `operator new(std::size_t)`

— `operator new(std::size_t, const std::nothrow_t&)`

— `operator new[](std::size_t)`

— `operator new[](std::size_t, const std::nothrow_t&)`

— `operator delete(void*)`

— `operator delete(void*, const std::nothrow_t&)`

— `operator delete[](void*)`

— `operator delete[](void*, const std::nothrow_t&)`

184) The function signatures declared in `<cuchar>`, `<cwchar>`, and `<cwctype>` are always reserved, notwithstanding the restrictions imposed in subclause 4.5.1 of Amendment 1 to the C Standard for these headers.

185) These types are `clock_t`, `div_t`, `FILE`, `fpos_t`, `lconv`, `ldiv_t`, `mbstate_t`, `ptrdiff_t`, `sig_atomic_t`, `size_t`, `time_t`, `tm`, `va_list`, `wctrans_t`, `wctype_t`, and `wint_t`.

— `operator delete(void*, std::size_t)`

— `operator delete(void*, std::size_t, const std::nothrow_t&)`

— `operator delete[](void*, std::size_t)`

— `operator delete[](void*, std::size_t, const std::nothrow_t&)`

1

The program's definitions are used instead of the default versions supplied by the implementation (18.159). Such replacement occurs prior to program startup (3.91, 3.1721). The program's definitions shall not be specified as `inline`. No diagnostic is required.

17.436.0.1 Handler functions [handler.functions]

1 The C++ standard library provides default versions of the following handler functions (Clause 18):

(1.1) — `unexpected_handler`

(1.2) — `terminate_handler`

2

A C++ program may install different handler functions during execution, by supplying a pointer to a function defined in the program or the library as an argument to (respectively):

— `set_new_handler`

— `set_unexpected`

— `set_terminate`

SEE ALSO: subclauses 18.223.1, Storage allocation errors, and 18.241, Exception handling.

1 A C++ program can get a pointer to the current handler function by calling the following functions:

(1.1) — `get_new_handler`

(1.2) — `get_unexpected`

(1.3) — `get_terminate`

2

Calling the `set_*` and `get_*` functions shall not incur a data race. A call to any of the `set_*` functions shall synchronize with subsequent calls to the same `set_*` function and to the corresponding `get_*` function.

17.438.0.1 Other functions [res.on.functions]

1

In certain cases (replacement functions, handler functions, operations on types used to instantiate standard library template components), the C++ standard library depends on components supplied by a C++ program. If these components do not meet their requirements, the Standard places no requirements on the implementation.

1

In particular, the effects are undefined in the following cases:

— for replacement functions (18.159.1), if the installed replacement function does not implement the semantics of the applicable *Required behavior:* paragraph.

— for handler functions (18.226.0.1, 18.247.2.1, D.62.1), if the installed handler function does not implement the semantics of the applicable *Required behavior:* paragraph

— for types used as template arguments when instantiating a template component, if the operations on the type do not implement the semantics of the applicable **Requirements** subclause (17.165.0.1, 23.2, 24.2, 26.2). Operations on such types can report a failure by throwing an exception unless otherwise specified.

— if any replacement function or handler function or destructor operation exits via an exception, unless specifically allowed in the applicable *Required behavior:* paragraph.

— if an incomplete type (3.2955) is used as a template argument when instantiating a template component, unless specifically allowed for that component.

17.513.0.1 Function arguments [res.on.arguments]

1

Each of the following applies to all arguments to functions defined in the C++ standard library, unless explicitly stated otherwise.

— If an argument to a function has an invalid value (such as a value outside the domain of the function or a pointer invalid for its intended use), the behavior is undefined.

— If a function argument is described as being an array, the pointer actually passed to the function shall have a value such that all address computations and accesses to objects (that would be valid if the pointer did point to the first element of such an array) are in fact valid.

— If a function argument binds to an rvalue reference parameter, the implementation may assume that this parameter is a unique reference to this argument. [*Note:* If the parameter is a generic parameter of the form `T&&` and an lvalue of type `A` is bound, the argument binds to an lvalue reference (14.2261.0.1) and thus is not covered by the previous sentence. — *end note*] [*Note:* If a program casts an lvalue to an xvalue while passing that lvalue to a library function (e.g. by calling the function with the argument `move(x)`), the program is effectively asking that function to treat that lvalue as a temporary. The implementation is free to optimize away aliasing checks which might be needed if the argument was an lvalue. — *end note*]

17.535.0.1 Shared objects and the library [res.on.objects]

1

The behavior of a program is undefined if calls to standard library functions from different threads may introduce a data race. The conditions under which this may occur are specified in 17.607.0.1. [*Note:* Modifying an object of a standard library type that is shared between threads risks undefined behavior unless objects of that type are explicitly specified as being sharable without data races or the user supplies a locking mechanism. — *end note*]

1 [*Note:* In particular, the program is required to ensure that completion of the constructor of any object of a class type defined in the standard library happens before any other member function invocation on that object and, unless otherwise specified, to ensure that completion of any member function invocation other than destruction on such an object happens before destruction of that object. This applies even to objects such as mutexes intended for thread synchronization. — *end note*]

17.539.0.1 Requires paragraph [res.on.required]

1

Violation of the preconditions specified in a function's *Requires:* paragraph results in undefined behavior unless the function's *Throws:* paragraph specifies throwing an exception when the precondition is violated.

17.544.1 Conforming implementations [conforming]

17.544.1.1 Overview [conforming.overview]

1 This section describes the constraints upon, and latitude of, implementations of the C++ standard library.

2 An implementation's use of headers is discussed in 17.544.1.2, its use of macros in 17.563.0.1, global functions in 17.565.0.1, member functions in 17.568.0.1, data race avoidance in 17.607.0.1, access specifiers in 17.674.0.1, class derivation in 17.674.0.2, and exceptions in 17.677.0.1.

17.544.1.2 Headers [res.on.headers]

1

A C++ header may include other C++ headers. A C++ header shall provide the declarations and definitions that appear in its synopsis. A C++ header shown in its synopsis as including other C++ headers shall provide the declarations and definitions that appear in the synopses of those other headers.

1

Certain types and macros are defined in more than one header. Every such entity shall be defined such that any header that defines it may be included after any other header that also defines it (3.91).

1

The C standard headers (D.9) shall include only their corresponding C++ standard header, as described in 17.48.0.1.

17.563.0.1 Restrictions on macro definitions [res.on.macro.definitions]

1 The names and global function signatures described in 17.37.1.1 are reserved to the implementation.

2

All object-like macros defined by the C standard library and described in this Clause as expanding to integral constant expressions are also suitable for use in `#if` preprocessing directives, unless explicitly stated otherwise.

17.565.0.1 Global and non-member functions [global.functions]

1

It is unspecified whether any global or non-member functions in the C++ standard library are defined as `inline` (7.94.1).

1 A call to a global or non-member function signature described in Clauses 18 through 30 and Annex D shall behave as if the implementation declared no additional global or non-member function signatures.[186]

2

An implementation shall not declare a global or non-member function signature with additional default arguments.

1

Unless otherwise specified, global and non-member functions in the standard library shall not use functions from another namespace which are found through *argument-dependent name lookup* (3.1092.1). [*Note:* The phrase "unless otherwise specified" is intended to allow argument-dependent lookup in cases like that of `ostream_iterator::operator=` (24.219.0.2):

Effects:

```
*out_stream << value;
```

186) A valid C++ program always calls the expected library global or non-member function. An implementation may also define additional global or non-member functions that would otherwise not be called by a valid C++ program.

```
if (delim != 0)
  *out_stream << delim;
return *this;
```

— *end note*]

17.568.0.1 Member functions [member.functions]

1

It is unspecified whether any member functions in the C++ standard library are defined as `inline` (7.94.1).

1

An implementation may declare additional non-virtual member function signatures within a class:

— by adding arguments with default values to a member function signature;[187] [*Note:* An implementation may not add arguments with default values to virtual, global, or non-member functions. — *end note*]

— by replacing a member function signature with default values by two or more member function signatures with equivalent behavior; and

— by adding a member function signature for a member function name.

1

A call to a member function signature described in the C++ standard library behaves as if the implementation declares no additional member function signatures.[188]

17.598.0.1 `constexpr` functions and constructors [constexpr.functions]

1

This standard explicitly requires that certain standard library functions are `constexpr` (7.328.1). An implementation shall not declare any standard library function signature as `constexpr` except for those where it is explicitly required. Within any header that provides any non-defining declarations of `constexpr` functions or constructors an implementation shall provide corresponding definitions.

17.602.0.1 Requirements for stable algorithms [algorithm.stable]

1

When the requirements for an algorithm state that it is "stable" without further elaboration, it means:

— For the *sort* algorithms the relative order of equivalent elements is preserved.

— For the *remove* and *copy* algorithms the relative order of the elements that are not removed is preserved.

— For the *merge* algorithms, for equivalent elements in the original two ranges, the elements from the first range (preserving their original order) precede the elements from the second range (preserving their original order).

17.604.0.1 Reentrancy [reentrancy]

1

Except where explicitly specified in this standard, it is implementation-defined which functions in the Standard C++ library may be recursively reentered.

187) Hence, the address of a member function of a class in the C++ standard library has an unspecified type.

188) A valid C++ program always calls the expected library member function, or one with equivalent behavior. An implementation may also define additional member functions that would otherwise not be called by a valid C++ program.

17.607.0.1 Data race avoidance [res.on.data.races]

1

This section specifies requirements that implementations shall meet to prevent data races (1.859). Every standard library function shall meet each requirement unless otherwise specified. Implementations may prevent data races in cases other than those specified below.

1

A C++ standard library function shall not directly or indirectly access objects (1.859) accessible by threads other than the current thread unless the objects are accessed directly or indirectly via the function's arguments, including `this`.

1

A C++ standard library function shall not directly or indirectly modify objects (1.859) accessible by threads other than the current thread unless the objects are accessed directly or indirectly via the function's non-const arguments, including `this`.

1

[*Note:* This means, for example, that implementations can't use a static object for internal purposes without synchronization because it could cause a data race even in programs that do not explicitly share objects between threads. — *end note*]

1

A C++ standard library function shall not access objects indirectly accessible via its arguments or via elements of its container arguments except by invoking functions required by its specification on those container elements.

1

Operations on iterators obtained by calling a standard library container or string member function may access the underlying container, but shall not modify it. [*Note:* In particular, container operations that invalidate iterators conflict with operations on iterators associated with that container. — *end note*]

1

Implementations may share their own internal objects between threads if the objects are not visible to users and are protected against data races.

1

Unless otherwise specified, C++ standard library functions shall perform all operations solely within the current thread if those operations have effects that are visible (1.859) to users.

1

[*Note:* This allows implementations to parallelize operations if there are no visible side effects. — *end note*]

17.674.0.1 Protection within classes [protection.within.classes]

1 It is unspecified whether any function signature or class described in Clauses 18 through 30 and Annex D is a `friend` of another class in the C++ standard library.

17.674.0.2 Derived classes [derivation]

1

An implementation may derive any class in the C++ standard library from a class with a name reserved to the implementation.

1 Certain classes defined in the C++ standard library are required to be derived from other classes in the C++ standard library. An implementation may derive such a class directly from the required base or indirectly

through a hierarchy of base classes with names reserved to the implementation.

2 In any case:

(2.1) — Every base class described as `virtual` shall be virtual;

(2.2) — Every base class described as non-`virtual` shall not be virtual;

(2.3) — Unless explicitly stated otherwise, types with distinct names shall be distinct types.[189]

17.677.0.1 Restrictions on exception handling [res.on.exception.handling]

1

Any of the functions defined in the C++ standard library can report a failure by throwing an exception of a type described in its **Throws:** paragraph. An implementation may strengthen the *exception-specification* for a non-virtual function by adding a non-throwing *noexcept-specification*.

1

A function may throw an object of a type not listed in its **Throws** clause if its type is derived from a type named in the **Throws** clause and would be caught by an exception handler for the base type.

1 Functions from the C standard library shall not throw exceptions[190] except when such a function calls a program-supplied function that throws an exception.[191]

2

Destructor operations defined in the C++ standard library shall not throw exceptions. Every destructor in the C++ standard library shall behave as if it had a non-throwing exception specification. Any other functions defined in the C++ standard library that do not have an *exception-specification* may throw implementation-defined exceptions unless otherwise specified.[192] An implementation may strengthen this implicit *exception-specification* by adding an explicit one.[193]

17.685.0.1 Restrictions on storage of pointers [res.on.pointer.storage]

1 Objects constructed by the standard library that may hold a user-supplied pointer value or an integer of type `std::intptr_t` shall store such values in a traceable pointer location (3.2532.0.1). [*Note:* Other libraries are strongly encouraged to do the same, since not doing so may result in accidental use of pointers that are not safely derived. Libraries that store pointers outside the user's address space should make it appear that they are stored and retrieved from a traceable pointer location. — *end note*]

17.685.0.2 Value of error codes [value.error.codes]

1

Certain functions in the C++ standard library report errors via a `std::error_code` (19.22.1.1) object. That object's `category()` member shall return `std::system_category()` for errors originating from the operating system, or a reference to an implementation-defined `error_category` object for errors originating elsewhere. The implementation shall define the possible values of `value()` for each of these error categories.

189) There is an implicit exception to this rule for types that are described as synonyms for basic integral types, such as `size_t` (18.2) and `streamoff` (27.56.2).

190) That is, the C library functions can all be treated as if they are marked `noexcept`. This allows implementations to make performance optimizations based on the absence of exceptions at runtime.

191) The functions `qsort()` and `bsearch()` (25.205) meet this condition.

192) In particular, they can report a failure to allocate storage by throwing an exception of type `bad_alloc`, or a class derived from `bad_alloc` (18.223.1.1). Library implementations should report errors by throwing exceptions of or derived from the standard exception classes (18.223.1.1, 18.241, 19.3).

193) That is, an implementation may provide an explicit *exception-specification* that defines the subset of "any" exceptions thrown by that function. This implies that the implementation may list implementation-defined types in such an *exception-specification*.

[*Example:* For operating systems that are based on POSIX, implementations are encouraged to define the `std::system_category()` values as identical to the POSIX `errno` values, with additional values as defined by the operating system's documentation. Implementations for operating systems that are not based on POSIX are encouraged to define values identical to the operating system's values. For errors that do not originate from the operating system, the implementation may provide enums for the associated values. — *end example*]

17.688.0.1 Moved-from state of library types [lib.types.movedfrom]

1

Objects of types defined in the C++ standard library may be moved from (12.1577). Move operations may be explicitly specified or implicitly generated. Unless otherwise specified, such moved-from objects shall be placed in a valid but unspecified state.

18 Language support library [language.support]

18.1 General [support.general]

1 This Clause describes the function signatures that are called implicitly, and the types of objects generated implicitly, during the execution of some C++ programs. It also describes the headers that declare these function signatures and define any related types.

2 The following subclauses describe common type definitions used throughout the library, characteristics of the predefined types, functions supporting start and termination of a C++ program, support for dynamic memory management, support for dynamic type identification, support for exception processing, support for initializer lists, and other runtime support, as summarized in Table 29.

Table 29 — Language support library summary

Subclause		Header(s)
18.2	Types	`<cstddef>`

18.41	Implementation properties	`<limits>`
		`<climits>`
		`<cfloat>`

18.114	Integer types	`<cstdint>`
18.127	Start and termination	`<cstdlib>`
18.159	Dynamic memory management	`<new>`
18.227	Type identification	`<typeinfo>`
18.241	Exception handling	`<exception>`
18.274	Initializer lists	`<initializer_list>`

18.295	Other runtime support	`<csignal>`
		`<csetjmp>`
		`<cstdalign>`
		`<cstdarg>`
		`<cstdbool>`
		`<cstdlib>`
		`<ctime>`

18.2 Types [support.types]

1 Table 30 describes the header `<cstddef>`.

Table 30 — Header `<cstddef>` synopsis

Type	Name(s)	
Macros:	`NULL`	`offsetof`
Types:	`ptrdiff_t`	`size_t`
	`max_align_t`	`nullptr_t`

2

The contents are the same as the Standard C library header `<stddef.h>`, with the following changes:

1

The macro `NULL` is an implementation-defined C++ null pointer constant in this International Standard (4.579).[194]

1

The macro `offsetof`(*type*, *member-designator*) accepts a restricted set of *type* arguments in this International Standard. If *type* is not a standard-layout class (Clause 9), the results are undefined.[195] The expression `offsetof`(*type*, *member-designator*) is never type-dependent (14.1390.0.1) and it is value-dependent (14.1430.0.1) if and only if *type* is dependent. The result of applying the `offsetof` macro to a field that is a static data member or a function member is undefined. No operation invoked by the `offsetof` macro shall throw an exception and `noexcept(offsetof(type, member-designator))` shall be `true`.

1 The type `ptrdiff_t` is an implementation-defined signed integer type that can hold the difference of two subscripts in an array object, as described in 5.2882.

2

The type `size_t` is an implementation-defined unsigned integer type that is large enough to contain the size in bytes of any object.

1

[*Note:* It is recommended that implementations choose types for `ptrdiff_t` and `size_t` whose integer conversion ranks (4.807) are no greater than that of `signed long int` unless a larger size is necessary to contain all the possible values. — *end note*]

1

The type `max_align_t` is a POD type whose alignment requirement is at least as great as that of every scalar type, and whose alignment requirement is supported in every context.

1

`nullptr_t` is defined as follows:

```
namespace std {
  typedef decltype(nullptr) nullptr_t;
}
```

The type for which `nullptr_t` is a synonym has the characteristics described in 3.3151.1 and 4.579. [*Note:* Although `nullptr`'s address cannot be taken, the address of another `nullptr_t` object that is an lvalue can be taken. — *end note*]

See also: Alignment (3.3924), Sizeof (5.2075.1), Additive operators (5.2882), Free store (12.988), and ISO C 7.1.6.

18.41 Implementation properties [support.limits]

18.41.1 In general [support.limits.general]

1 The headers `<limits>` (18.41.2), `<climits>`, and `<cfloat>` (18.110.1) supply characteristics of implementation-dependent arithmetic types (3.3151.1).

18.41.2 Numeric limits [limits]

18.41.2.1 Class template `numeric_limits` [limits.numeric]

1

194) Possible definitions include `0` and `0L`, but not `(void*)0`.

195) Note that `offsetof` is required to work as specified even if unary `operator&` is overloaded for any of the types involved.

The `numeric_limits` class template provides a C++ program with information about various properties of the implementation's representation of the arithmetic types.

1

Specializations shall be provided for each arithmetic type, both floating point and integer, including `bool`. The member `is_specialized` shall be `true` for all such specializations of `numeric_limits`.

1

For all members declared `static constexpr` in the `numeric_limits` template, specializations shall define these values in such a way that they are usable as constant expressions.

1

Non-arithmetic standard types, such as `complex<T>` (26.33.2), shall not have specializations.

18.53.0.1 Header `<limits>` synopsis [limits.syn]

```
namespace std {
  template<class T> class numeric_limits;
  enum float_round_style;
  enum float_denorm_style;

  template<> class numeric_limits<bool>;

  template<> class numeric_limits<char>;
  template<> class numeric_limits<signed char>;
  template<> class numeric_limits<unsigned char>;
  template<> class numeric_limits<char16_t>;
  template<> class numeric_limits<char32_t>;
  template<> class numeric_limits<wchar_t>;

  template<> class numeric_limits<short>;
  template<> class numeric_limits<int>;
  template<> class numeric_limits<long>;
  template<> class numeric_limits<long long>;
  template<> class numeric_limits<unsigned short>;
  template<> class numeric_limits<unsigned int>;
  template<> class numeric_limits<unsigned long>;
  template<> class numeric_limits<unsigned long long>;

  template<> class numeric_limits<float>;
  template<> class numeric_limits<double>;
  template<> class numeric_limits<long double>;
}
```

18.53.0.2 Class template `numeric_limits` [numeric.limits]

```
namespace std {
  template<class T> class numeric_limits {
  public:
    static constexpr bool is_specialized = false;
    static constexpr T min() noexcept { return T(); }
    static constexpr T max() noexcept { return T(); }
    static constexpr T lowest() noexcept { return T(); }

    static constexpr int  digits = 0;
    static constexpr int  digits10 = 0;
```

```
    static constexpr int  max_digits10 = 0;
    static constexpr bool is_signed = false;
    static constexpr bool is_integer = false;
    static constexpr bool is_exact = false;
    static constexpr int  radix = 0;
    static constexpr T epsilon() noexcept { return T(); }
    static constexpr T round_error() noexcept { return T(); }

    static constexpr int  min_exponent = 0;
    static constexpr int  min_exponent10 = 0;
    static constexpr int  max_exponent = 0;
    static constexpr int  max_exponent10 = 0;

    static constexpr bool has_infinity = false;
    static constexpr bool has_quiet_NaN = false;
    static constexpr bool has_signaling_NaN = false;
    static constexpr float_denorm_style has_denorm = denorm_absent;
    static constexpr bool has_denorm_loss = false;
    static constexpr T infinity() noexcept { return T(); }
    static constexpr T quiet_NaN() noexcept { return T(); }
    static constexpr T signaling_NaN() noexcept { return T(); }
    static constexpr T denorm_min() noexcept { return T(); }

    static constexpr bool is_iec559 = false;
    static constexpr bool is_bounded = false;
    static constexpr bool is_modulo = false;

    static constexpr bool traps = false;
    static constexpr bool tinyness_before = false;
    static constexpr float_round_style round_style = round_toward_zero;
  };

  template<class T> class numeric_limits<const T>;
  template<class T> class numeric_limits<volatile T>;
  template<class T> class numeric_limits<const volatile T>;
}
```

1

The default `numeric_limits<T>` template shall have all members, but with 0 or `false` values.

1

The value of each member of a specialization of `numeric_limits` on a *cv*-qualified type `cv T` shall be equal to the value of the corresponding member of the specialization on the unqualified type `T`.

18.57.0.1 `numeric_limits` members [numeric.limits.members]

`static constexpr T min() noexcept;`

1 Minimum finite value.[196]

2

For floating types with denormalization, returns the minimum positive normalized value.

1

196) Equivalent to `CHAR_MIN`, `SHRT_MIN`, `FLT_MIN`, `DBL_MIN`, etc.

Meaningful for all specializations in which `is_bounded != false`, or `is_bounded == false && is_signed == false`.

`static constexpr T max() noexcept;`

1 Maximum finite value.[197]

2

Meaningful for all specializations in which `is_bounded != false`.

`static constexpr T lowest() noexcept;`

1

A finite value `x` such that there is no other finite value `y` where `y < x`.[198]

1

Meaningful for all specializations in which `is_bounded != false`.

`static constexpr int digits;`

1

Number of `radix` digits that can be represented without change.

1

For integer types, the number of non-sign bits in the representation.

1 For floating point types, the number of `radix` digits in the mantissa.[199]

`static constexpr int digits10;`

1 Number of base 10 digits that can be represented without change.[200]

2 Meaningful for all specializations in which `is_bounded != false`.

`static constexpr int max_digits10;`

3 Number of base 10 digits required to ensure that values which differ are always differentiated.

4

Meaningful for all floating point types.

`static constexpr bool is_signed;`

1 True if the type is signed.

2 Meaningful for all specializations.

`static constexpr bool is_integer;`

3 True if the type is integer.

4

Meaningful for all specializations.

`static constexpr bool is_exact;`

197) Equivalent to `CHAR_MAX`, `SHRT_MAX`, `FLT_MAX`, `DBL_MAX`, etc.

198) `lowest()` is necessary because not all floating-point representations have a smallest (most negative) value that is the negative of the largest (most positive) finite value.

199) Equivalent to `FLT_MANT_DIG`, `DBL_MANT_DIG`, `LDBL_MANT_DIG`.

200) Equivalent to `FLT_DIG`, `DBL_DIG`, `LDBL_DIG`.

1

True if the type uses an exact representation. All integer types are exact, but not all exact types are integer. For example, rational and fixed-exponent representations are exact but not integer.

1 Meaningful for all specializations.

```
static constexpr int radix;
```

2 For floating types, specifies the base or radix of the exponent representation (often 2).[201]

3 For integer types, specifies the base of the representation.[202]

4 Meaningful for all specializations.

```
static constexpr T epsilon() noexcept;
```

5 Machine epsilon: the difference between 1 and the least value greater than 1 that is representable.[203]

6 Meaningful for all floating point types.

```
static constexpr T round_error() noexcept;
```

7 Measure of the maximum rounding error.[204]

```
static constexpr int  min_exponent;
```

8 Minimum negative integer such that `radix` raised to the power of one less than that integer is a normalized floating point number.[205]

9 Meaningful for all floating point types.

```
static constexpr int  min_exponent10;
```

10 Minimum negative integer such that 10 raised to that power is in the range of normalized floating point numbers.[206]

11 Meaningful for all floating point types.

```
static constexpr int  max_exponent;
```

12 Maximum positive integer such that `radix` raised to the power one less than that integer is a representable finite floating point number.[207]

13 Meaningful for all floating point types.

```
static constexpr int  max_exponent10;
```

14 Maximum positive integer such that 10 raised to that power is in the range of representable finite floating point numbers.[208]

15 Meaningful for all floating point types.

```
static constexpr bool has_infinity;
```

201) Equivalent to `FLT_RADIX`.
202) Distinguishes types with bases other than 2 (e.g. BCD).
203) Equivalent to `FLT_EPSILON`, `DBL_EPSILON`, `LDBL_EPSILON`.
204) Rounding error is described in ISO/IEC 10967-1 Language independent arithmetic - Part 1 Section 5.2.8 and Annex A Rationale Section A.5.2.8 - Rounding constants.
205) Equivalent to `FLT_MIN_EXP`, `DBL_MIN_EXP`, `LDBL_MIN_EXP`.
206) Equivalent to `FLT_MIN_10_EXP`, `DBL_MIN_10_EXP`, `LDBL_MIN_10_EXP`.
207) Equivalent to `FLT_MAX_EXP`, `DBL_MAX_EXP`, `LDBL_MAX_EXP`.
208) Equivalent to `FLT_MAX_10_EXP`, `DBL_MAX_10_EXP`, `LDBL_MAX_10_EXP`.

16 True if the type has a representation for positive infinity.

17 Meaningful for all floating point types.

18

Shall be `true` for all specializations in which `is_iec559 != false`.

```
static constexpr bool has_quiet_NaN;
```

1

True if the type has a representation for a quiet (non-signaling) "Not a Number."[209]

1 Meaningful for all floating point types.

2 Shall be `true` for all specializations in which `is_iec559 != false`.

```
static constexpr bool has_signaling_NaN;
```

3 True if the type has a representation for a signaling "Not a Number."[210]

4 Meaningful for all floating point types.

5 Shall be `true` for all specializations in which `is_iec559 != false`.

```
static constexpr float_denorm_style has_denorm;
```

6 `denorm_present` if the type allows denormalized values (variable number of exponent bits)[211], `denorm_absent` if the type does not allow denormalized values, and `denorm_indeterminate` if it is indeterminate at compile time whether the type allows denormalized values.

7 Meaningful for all floating point types.

```
static constexpr bool has_denorm_loss;
```

8

True if loss of accuracy is detected as a denormalization loss, rather than as an inexact result.[212]

```
static constexpr T infinity() noexcept;
```

1

Representation of positive infinity, if available.[213]

1

Meaningful for all specializations for which `has_infinity != false`. Required in specializations for which `is_iec559 != false`.

```
static constexpr T quiet_NaN() noexcept;
```

1 Representation of a quiet "Not a Number," if available.[214]

2 Meaningful for all specializations for which `has_quiet_NaN != false`. Required in specializations for which `is_iec559 != false`.

```
static constexpr T signaling_NaN() noexcept;
```

209) Required by LIA-1.
210) Required by LIA-1.
211) Required by LIA-1.
212) See IEC 559.
213) Required by LIA-1.
214) Required by LIA-1.

3 Representation of a signaling "Not a Number," if available.[215]

4 Meaningful for all specializations for which `has_signaling_NaN != false`. Required in specializations for which `is_iec559 != false`.

`static constexpr T denorm_min() noexcept;`

5 Minimum positive denormalized value.[216]

6 Meaningful for all floating point types.

7

In specializations for which `has_denorm == false`, returns the minimum positive normalized value.

`static constexpr bool is_iec559;`

1

True if and only if the type adheres to IEC 559 standard.[217]

1 Meaningful for all floating point types.

`static constexpr bool is_bounded;`

2 True if the set of values representable by the type is finite.[218] [*Note:* All fundamental types (3.3151.1) are bounded. This member would be false for arbitrary precision types. — *end note*]

3 Meaningful for all specializations.

`static constexpr bool is_modulo;`

4

True if the type is modulo.[219] A type is modulo if, for any operation involving `+`, `-`, or `*` on values of that type whose result would fall outside the range `[min(),max()]`, the value returned differs from the true value by an integer multiple of `max() - min() + 1`.

1

On most machines, this is `false` for floating types, `true` for unsigned integers, and `true` for signed integers.

1 Meaningful for all specializations.

`static constexpr bool traps;`

2 `true` if, at program startup, there exists a value of the type that would cause an arithmetic operation using that value to trap.[220]

3 Meaningful for all specializations.

`static constexpr bool tinyness_before;`

4 `true` if tinyness is detected before rounding.[221]

5 Meaningful for all floating point types.

215) Required by LIA-1.
216) Required by LIA-1.
217) International Electrotechnical Commission standard 559 is the same as IEEE 754.
218) Required by LIA-1.
219) Required by LIA-1.
220) Required by LIA-1.
221) Refer to IEC 559. Required by LIA-1.

```
static constexpr float_round_style round_style;
```

6 The rounding style for the type.[222]

7 Meaningful for all floating point types. Specializations for integer types shall return `round_toward_zero`.

18.107.0.1 Type `float_round_style` [round.style]

```
namespace std {
  enum float_round_style {
    round_indeterminate       = -1,
    round_toward_zero         =  0,
    round_to_nearest          =  1,
    round_toward_infinity     =  2,
    round_toward_neg_infinity =  3
  };
}
```

1 The rounding mode for floating point arithmetic is characterized by the values:

(1.1) — `round_indeterminate` if the rounding style is indeterminable

(1.2) — `round_toward_zero` if the rounding style is toward zero

(1.3) — `round_to_nearest` if the rounding style is to the nearest representable value

(1.4) — `round_toward_infinity` if the rounding style is toward infinity

(1.5) — `round_toward_neg_infinity` if the rounding style is toward negative infinity

18.107.0.2 Type `float_denorm_style` [denorm.style]

```
namespace std {
  enum float_denorm_style {
    denorm_indeterminate = -1,
    denorm_absent = 0,
    denorm_present = 1
  };
}
```

1 The presence or absence of denormalization (variable number of exponent bits) is characterized by the values:

(1.1) — `denorm_indeterminate` if it cannot be determined whether or not the type allows denormalized values

(1.2) — `denorm_absent` if the type does not allow denormalized values

(1.3) — `denorm_present` if the type does allow denormalized values

18.107.0.3 `numeric_limits` specializations [numeric.special]

1

All members shall be provided for all specializations. However, many values are only required to be meaningful under certain conditions (for example, `epsilon()` is only meaningful if `is_integer` is `false`). Any value that is not "meaningful" shall be set to 0 or `false`.

1 [*Example:*

222) Equivalent to `FLT_ROUNDS`. Required by LIA-1.

```
namespace std {
  template<> class numeric_limits<float> {
  public:
    static constexpr bool is_specialized = true;

    inline static constexpr float min() noexcept { return 1.17549435E-38F; }
    inline static constexpr float max() noexcept { return 3.40282347E+38F; }
    inline static constexpr float lowest() noexcept { return -3.40282347E+38F; }

    static constexpr int digits   = 24;
    static constexpr int digits10 =  6;
    static constexpr int max_digits10 =  9;

    static constexpr bool is_signed  = true;
    static constexpr bool is_integer = false;
    static constexpr bool is_exact   = false;

    static constexpr int radix = 2;
    inline static constexpr float epsilon() noexcept     { return 1.19209290E-07F; }
    inline static constexpr float round_error() noexcept { return 0.5F; }

    static constexpr int min_exponent   = -125;
    static constexpr int min_exponent10 = - 37;
    static constexpr int max_exponent   = +128;
    static constexpr int max_exponent10 = + 38;

    static constexpr bool has_infinity             = true;
    static constexpr bool has_quiet_NaN            = true;
    static constexpr bool has_signaling_NaN        = true;
    static constexpr float_denorm_style has_denorm = denorm_absent;
    static constexpr bool has_denorm_loss          = false;

    inline static constexpr float infinity()      noexcept { return value; }
    inline static constexpr float quiet_NaN()     noexcept { return value; }
    inline static constexpr float signaling_NaN() noexcept { return value; }
    inline static constexpr float denorm_min()    noexcept { return min(); }

    static constexpr bool is_iec559  = true;
    static constexpr bool is_bounded = true;
    static constexpr bool is_modulo  = false;
    static constexpr bool traps      = true;
    static constexpr bool tinyness_before = true;

    static constexpr float_round_style round_style = round_to_nearest;
  };
}
```

— *end example*]

2 The specialization for `bool` shall be provided as follows:

```
namespace std {
   template<> class numeric_limits<bool> {
   public:
     static constexpr bool is_specialized = true;
     static constexpr bool min() noexcept { return false; }
```

```
    static constexpr bool max() noexcept { return true; }
    static constexpr bool lowest() noexcept { return false; }

    static constexpr int  digits = 1;
    static constexpr int  digits10 = 0;
    static constexpr int  max_digits10 = 0;

    static constexpr bool is_signed = false;
    static constexpr bool is_integer = true;
    static constexpr bool is_exact = true;
    static constexpr int  radix = 2;
    static constexpr bool epsilon() noexcept { return 0; }
    static constexpr bool round_error() noexcept { return 0; }

    static constexpr int  min_exponent = 0;
    static constexpr int  min_exponent10 = 0;
    static constexpr int  max_exponent = 0;
    static constexpr int  max_exponent10 = 0;

    static constexpr bool has_infinity = false;
    static constexpr bool has_quiet_NaN = false;
    static constexpr bool has_signaling_NaN = false;
    static constexpr float_denorm_style has_denorm = denorm_absent;
    static constexpr bool has_denorm_loss = false;
    static constexpr bool infinity() noexcept { return 0; }
    static constexpr bool quiet_NaN() noexcept { return 0; }
    static constexpr bool signaling_NaN() noexcept { return 0; }
    static constexpr bool denorm_min() noexcept { return 0; }

    static constexpr bool is_iec559 = false;
    static constexpr bool is_bounded = true;
    static constexpr bool is_modulo = false;

    static constexpr bool traps = false;
    static constexpr bool tinyness_before = false;
    static constexpr float_round_style round_style = round_toward_zero;
  };
}
```

18.110.1 C library [c.limits]

1 Table 31 describes the header `<climits>`.

Table 31 — Header `<climits>` synopsis

Type	Name(s)				
Values:					
CHAR_BIT	INT_MAX	LONG_MAX	SCHAR_MIN	SHRT_MIN	ULLONG_MAX
CHAR_MAX	LLONG_MAX	LONG_MIN	SCHAR_MAX	UCHAR_MAX	ULONG_MAX
CHAR_MIN	LLONG_MIN	MB_LEN_MAX	SHRT_MAX	UINT_MAX	USHRT_MAX
INT_MIN					

2

The contents are the same as the Standard C library header `<limits.h>`. [*Note:* The types of the constants defined by macros in `<climits>` are not required to match the types to which the macros refer. — *end note*]

1 Table 32 describes the header `<cfloat>`.

Table 32 — Header `<cfloat>` synopsis

Type	Name(s)		
Values:			
DBL_DIG	DBL_MIN_EXP	FLT_MAX_EXP	LDBL_MANT_DIG
DBL_EPSILON	DECIMAL_DIG	FLT_MIN	LDBL_MAX_10_EXP
DBL_MANT_DIG	FLT_DIG	FLT_MIN_10_EXP	LDBL_MAX_EXP
DBL_MAX	FLT_EPSILON	FLT_MIN_EXP	LDBL_MAX
DBL_MAX_10_EXP	FLT_EVAL_METHOD	FLT_RADIX	LDBL_MIN
DBL_MAX_EXP	FLT_MANT_DIG	FLT_ROUNDS	LDBL_MIN_10_EXP
DBL_MIN	FLT_MAX	LDBL_DIG	LDBL_MIN_EXP
DBL_MIN_10_EXP	FLT_MAX_10_EXP	LDBL_EPSILON	

2

The contents are the same as the Standard C library header `<float.h>`.

See also: ISO C 7.1.5, 5.2.4.2.2, 5.2.4.2.1.

18.114 Integer types [cstdint]

18.114.1 Header `<cstdint>` synopsis [cstdint.syn]

```
namespace std {
  typedef signed integer type int8_t;   // optional
  typedef signed integer type int16_t;  // optional
  typedef signed integer type int32_t;  // optional
  typedef signed integer type int64_t;  // optional

  typedef signed integer type int_fast8_t;
  typedef signed integer type int_fast16_t;
  typedef signed integer type int_fast32_t;
  typedef signed integer type int_fast64_t;

  typedef signed integer type int_least8_t;
  typedef signed integer type int_least16_t;
  typedef signed integer type int_least32_t;
  typedef signed integer type int_least64_t;

  typedef signed integer type intmax_t;
  typedef signed integer type intptr_t;         // optional

  typedef unsigned integer type uint8_t;        // optional
  typedef unsigned integer type uint16_t;       // optional
  typedef unsigned integer type uint32_t;       // optional
  typedef unsigned integer type uint64_t;       // optional

  typedef unsigned integer type uint_fast8_t;
  typedef unsigned integer type uint_fast16_t;
  typedef unsigned integer type uint_fast32_t;
  typedef unsigned integer type uint_fast64_t;
```

```
  typedef unsigned integer type uint_least8_t;
  typedef unsigned integer type uint_least16_t;
  typedef unsigned integer type uint_least32_t;
  typedef unsigned integer type uint_least64_t;

  typedef unsigned integer type uintmax_t;
  typedef unsigned integer type uintptr_t;      // optional
} // namespace std
```

1

The header also defines numerous macros of the form:

```
INT_[FAST LEAST]{8 16 32 64}_MIN
[U]INT_[FAST LEAST]{8 16 32 64}_MAX
INT{MAX PTR}_MIN
[U]INT{MAX PTR}_MAX
{PTRDIFF SIG_ATOMIC WCHAR WINT}{_MAX _MIN}
SIZE_MAX
```

plus function macros of the form:

```
[U]INT{8 16 32 64 MAX}_C
```

1

The header defines all functions, types, and macros the same as 7.18 in the C standard. [*Note:* The macros defined by `<cstdint>` are provided unconditionally. In particular, the symbols `__STDC_LIMIT_MACROS` and `__STDC_CONSTANT_MACROS` (mentioned in footnotes 219, 220, and 222 in the C standard) play no role in C++. — *end note*]

18.127 Start and termination [support.start.term]

1 Table 33 describes some of the contents of the header `<cstdlib>`.

Table 33 — Header `<cstdlib>` synopsis

Type	Name(s)		
Macros:	`EXIT_FAILURE`	`EXIT_SUCCESS`	
Functions:	`_Exit`	`abort`	`atexit`
	`at_quick_exit`	`exit`	`quick_exit`

2 The contents are the same as the Standard C library header `<stdlib.h>`, with the following changes:

```
[[noreturn]] void _Exit(int status) noexcept;
```

3

The function `_Exit(int status)` has additional behavior in this International Standard:

— The program is terminated without executing destructors for objects of automatic, thread, or static storage duration and without calling functions passed to `atexit()` (3.2200.1).

```
[[noreturn]] void abort(void) noexcept;
```

1

The function `abort()` has additional behavior in this International Standard:

— The program is terminated without executing destructors for objects of automatic, thread, or static storage duration and without calling functions passed to `atexit()` (3.2200.1).

```
extern "C" int atexit(void (*f)(void)) noexcept;
extern "C++" int atexit(void (*f)(void)) noexcept;
```

1 *Effects:* The `atexit()` functions register the function pointed to by `f` to be called without arguments at normal program termination. It is unspecified whether a call to `atexit()` that does not happen before (1.859) a call to `exit()` will succeed. [*Note:* The `atexit()` functions do not introduce a data race (17.607.0.1). — *end note*]

1 *Implementation limits:* The implementation shall support the registration of at least 32 functions.

1 *Returns:* The `atexit()` function returns zero if the registration succeeds, non-zero if it fails.

```
[[noreturn]] void exit(int status)
```

2 The function `exit()` has additional behavior in this International Standard:

— First, objects with thread storage duration and associated with the current thread are destroyed. Next, objects with static storage duration are destroyed and functions registered by calling `atexit` are called.[223] See 3.2200.1 for the order of destructions and calls. (Automatic objects are not destroyed as a result of calling `exit()`.)[224]

If control leaves a registered function called by `exit` because the function does not provide a handler for a thrown exception, `std::terminate()` shall be called (15.308.1).

— Next, all open C streams (as mediated by the function signatures declared in `<cstdio>`) with unwritten buffered data are flushed, all open C streams are closed, and all files created by calling `tmpfile()` are removed.

— Finally, control is returned to the host environment. If `status` is zero or `EXIT_SUCCESS`, an implementation-defined form of the status *successful termination* is returned. If `status` is `EXIT_FAILURE`, an implementation-defined form of the status *unsuccessful termination* is returned. Otherwise the status returned is implementation-defined.[225]

```
extern "C" int at_quick_exit(void (*f)(void)) noexcept;
extern "C++" int at_quick_exit(void (*f)(void)) noexcept;
```

1 *Effects:* The `at_quick_exit()` functions register the function pointed to by `f` to be called without arguments when `quick_exit` is called. It is unspecified whether a call to `at_quick_exit()` that does not happen before (1.859) all calls to `quick_exit` will succeed. [*Note:* The `at_quick_exit()` functions do not introduce a data race (17.607.0.1). — *end note*] [*Note:* The order of registration may be indeterminate if `at_quick_exit` was called from more than one thread. — *end note*] [*Note:* The `at_quick_exit` registrations are distinct from the `atexit` registrations, and applications may need to call both registration functions with the same argument. — *end note*]

2 *Implementation limits:* The implementation shall support the registration of at least 32 functions.

3 *Returns:* Zero if the registration succeeds, non-zero if it fails.

```
[[noreturn]] void quick_exit(int status) noexcept;
```

223) A function is called for every time it is registered.

224) Objects with automatic storage duration are all destroyed in a program whose function `main()` contains no automatic objects and executes the call to `exit()`. Control can be transferred directly to such a `main()` by throwing an exception that is caught in `main()`.

225) The macros `EXIT_FAILURE` and `EXIT_SUCCESS` are defined in `<cstdlib>`.

4

Effects: Functions registered by calls to `at_quick_exit` are called in the reverse order of their registration, except that a function shall be called after any previously registered functions that had already been called at the time it was registered. Objects shall not be destroyed as a result of calling `quick_exit`. If control leaves a registered function called by `quick_exit` because the function does not provide a handler for a thrown exception, `std::terminate()` shall be called. [*Note:* `at_quick_exit` may call a registered function from a different thread than the one that registered it, so registered functions should not rely on the identity of objects with thread storage duration. — *end note*] After calling registered functions, `quick_exit` shall call `_Exit(status)`. [*Note:* The standard file buffers are not flushed. SEE: ISO C 7.20.4.4. — *end note*]

SEE ALSO: 3.1721, 3.2200.1, ISO C 7.10.4.

18.159 Dynamic memory management [support.dynamic]

1 The header `<new>` defines several functions that manage the allocation of dynamic storage in a program. It also defines components for reporting storage management errors.

Header `<new>` synopsis

```
namespace std {
  class bad_alloc;
  class bad_array_new_length;
  struct nothrow_t {};
  extern const nothrow_t nothrow;
  typedef void (*new_handler)();
  new_handler get_new_handler() noexcept;
  new_handler set_new_handler(new_handler new_p) noexcept;
}

void* operator new(std::size_t size);
void* operator new(std::size_t size, const std::nothrow_t&) noexcept;
void  operator delete(void* ptr) noexcept;
void  operator delete(void* ptr, const std::nothrow_t&) noexcept;
void  operator delete(void* ptr, std::size_t size) noexcept;
void  operator delete(void* ptr, std::size_t size,
                      const std::nothrow_t&) noexcept;
void* operator new[](std::size_t size);
void* operator new[](std::size_t size, const std::nothrow_t&) noexcept;
void  operator delete[](void* ptr) noexcept;
void  operator delete[](void* ptr, const std::nothrow_t&) noexcept;
void  operator delete[](void* ptr, std::size_t size) noexcept;
void  operator delete[](void* ptr, std::size_t size,
                        const std::nothrow_t&) noexcept;

void* operator new  (std::size_t size, void* ptr) noexcept;
void* operator new[](std::size_t size, void* ptr) noexcept;
void  operator delete  (void* ptr, void*) noexcept;
void  operator delete[](void* ptr, void*) noexcept;
```

SEE ALSO: 1.277, 3.2331.1, 5.2221.1, 5.2455.1, 12.988, 20.82.

18.159.1 Storage allocation and deallocation [new.delete]

1 Except where otherwise specified, the provisions of (3.2331.1) apply to the library versions of `operator new` and `operator delete`.

18.159.1.1 Single-object forms [new.delete.single]

```
void* operator new(std::size_t size);
```

1 *Effects:* The *allocation function* (3.2361.0.1) called by a *new-expression* (5.2221.1) to allocate `size` bytes of storage suitably aligned to represent any object of that size.

1 *Replaceable:* a C++ program may define a function with this function signature that displaces the default version defined by the C++ standard library.

1 *Required behavior:* Return a non-null pointer to suitably aligned storage (3.2331.1), or else throw a `bad_alloc` exception. This requirement is binding on a replacement version of this function.

1 *Default behavior:*

— Executes a loop: Within the loop, the function first attempts to allocate the requested storage. Whether the attempt involves a call to the Standard C library function `malloc` is unspecified.

— Returns a pointer to the allocated storage if the attempt is successful. Otherwise, if the current `new_handler` (18.226.0.3) is a null pointer value, throws `bad_alloc`.

— Otherwise, the function calls the current `new_handler` function (18.226.0.1). If the called function returns, the loop repeats.

— The loop terminates when an attempt to allocate the requested storage is successful or when a called `new_handler` function does not return.

```
void* operator new(std::size_t size, const std::nothrow_t&) noexcept;
```

1 *Effects:* Same as above, except that it is called by a placement version of a *new-expression* when a C++ program prefers a null pointer result as an error indication, instead of a `bad_alloc` exception.

1 *Replaceable:* a C++ program may define a function with this function signature that displaces the default version defined by the C++ standard library.

2 *Required behavior:* Return a non-null pointer to suitably aligned storage (3.2331.1), or else return a null pointer. This nothrow version of `operator new` returns a pointer obtained as if acquired from the (possibly replaced) ordinary version. This requirement is binding on a replacement version of this function.

1 *Default behavior:* Calls `operator new(size)`. If the call returns normally, returns the result of that call. Otherwise, returns a null pointer.

2 [*Example:*

```
T* p1 = new T;                  // throws bad_alloc if it fails
T* p2 = new(nothrow) T;         // returns nullptr if it fails
```

— *end example*]

```
void operator delete(void* ptr) noexcept;
void operator delete(void* ptr, std::size_t size) noexcept;
```

1 *Effects:* The *deallocation function* (3.2428.0.1) called by a *delete-expression* to render the value of `ptr` invalid.

2 *Replaceable:* a C++ program may define a function with signature `void operator delete(void* ptr) noexcept` that displaces the default version defined by the C++ standard library. If this function (without `size` parameter) is defined, the program should also define `void operator delete(void* ptr, std::size_t size) noexcept`. If this function with `size` parameter is defined, the program shall also define the version without the `size` parameter. [*Note:* The default behavior below may change in the future, which will require replacing both deallocation functions when replacing the allocation function. — *end note*]

1 *Requires:* *ptr* shall be a null pointer or its value shall be a value returned by an earlier call to the (possibly replaced) `operator new(std::size_t)` or `operator new(std::size_t,const std::nothrow_t&)` which has not been invalidated by an intervening call to `operator delete(void*)`.

1 *Requires:* If an implementation has strict pointer safety (3.2532.0.1) then `ptr` shall be a safely-derived pointer.

2 *Requires:* If present, the `std::size_t size` argument shall equal the size argument passed to the allocation function that returned `ptr`.

3 *Required behavior:* Calls to `operator delete(void* ptr, std::size_t size)` may be changed to calls to `operator delete(void* ptr)` without affecting memory allocation. [*Note:* A conforming implementation is for `operator delete(void* ptr, std::size_t size)` to simply call `operator delete(ptr)`. — *end note*]

4 *Default behavior:* the function `operator delete(void* ptr, std::size_t size)` calls `operator delete(ptr)`. [*Note:* See the note in the above *Replaceable* paragraph. — *end note*]

5 *Default behavior:* If `ptr` is null, does nothing. Otherwise, reclaims the storage allocated by the earlier call to `operator new`.

1 *Remarks:* It is unspecified under what conditions part or all of such reclaimed storage will be allocated by subsequent calls to `operator new` or any of `calloc`, `malloc`, or `realloc`, declared in `<cstdlib>`.

```
void operator delete(void* ptr, const std::nothrow_t&) noexcept;
void operator delete(void* ptr, std::size_t size, const std::nothrow_t&) noexcept;
```

1 *Effects:* The *deallocation function* (3.2428.0.1) called by the implementation to render the value of `ptr` invalid when the constructor invoked from a nothrow placement version of the *new-expression* throws an exception.

2 *Replaceable:* a C++ program may define a function with signature `void operator delete(void* ptr, const std::nothrow_t&) noexcept` that displaces the default version defined by the C++ standard library. If this function (without `size` parameter) is defined, the program should also define `void operator delete(void* ptr, std::size_t size, const std::nothrow_t&) noexcept`. If this function with `size` parameter is defined, the program shall also define the version without the `size` parameter. [*Note:* The default behavior below may change in the future, which will require replacing both deallocation functions when replacing the allocation function. — *end note*]

3 *Requires:* If an implementation has strict pointer safety (3.2532.0.1) then `ptr` shall be a safely-derived pointer.

4 *Requires:* If present, the `std::size_t size` argument must equal the size argument passed to the allocation function that returned `ptr`.

5 *Required behavior:* Calls to `operator delete(void* ptr, std::size_t size, const std::nothrow_t&)` may be changed to calls to `operator delete(void* ptr, const std::nothrow_t&)` without affecting memory allocation. [*Note:* A conforming implementation is for `operator delete(void* ptr, std::size_t size, const std::nothrow_t&)` to simply call `operator delete(void* ptr, const std::nothrow_t&)`. — *end note*]

6 *Default behavior:* `operator delete(void* ptr, std::size_t size, const std::nothrow_t&)` calls `operator delete(ptr, std::nothrow)`, and `operator delete(void* ptr, const std::nothrow_t&)` calls `operator delete(ptr)`.

18.196.0.1 Array forms [new.delete.array]

```
void* operator new[](std::size_t size);
```

1 *Effects:* The *allocation function* (3.2361.0.1) called by the array form of a *new-expression* (5.2221.1) to allocate `size` bytes of storage suitably aligned to represent any array object of that size or smaller.[226]

1 *Replaceable:* a C++ program can define a function with this function signature that displaces the default version defined by the C++ standard library.

2 *Required behavior:* Same as for `operator new(std::size_t)`. This requirement is binding on a replacement version of this function.

1 *Default behavior:* Returns `operator new(size)`.

```
void* operator new[](std::size_t size, const std::nothrow_t&) noexcept;
```

1 *Effects:* Same as above, except that it is called by a placement version of a *new-expression* when a C++ program prefers a null pointer result as an error indication, instead of a `bad_alloc` exception.

2 *Replaceable:* a C++ program can define a function with this function signature that displaces the default version defined by the C++ standard library.

3 *Required behavior:* Return a non-null pointer to suitably aligned storage (3.2331.1), or return a null pointer. This requirement is binding on a replacement version of this function.

4 *Default behavior:* Calls `operator new[](size)`. If the call returns normally, returns the result of that call. Otherwise, returns a null pointer.

```
void operator delete[](void* ptr) noexcept;
void operator delete[](void* ptr, std::size_t size) noexcept;
```

5 *Effects:* The *deallocation function* (3.2428.0.1) called by the array form of a *delete-expression* to render the value of `ptr` invalid.

1 *Replaceable:* a C++ program can define a function with signature `void operator delete[](void* ptr) noexcept` that displaces the default version defined by the C++ standard library. If this function

226) It is not the direct responsibility of `operator new[](std::size_t)` or `operator delete[](void*)` to note the repetition count or element size of the array. Those operations are performed elsewhere in the array `new` and `delete` expressions. The array `new` expression, may, however, increase the `size` argument to `operator new[](std::size_t)` to obtain space to store supplemental information.

(without `size` parameter) is defined, the program should also define `void operator delete[](void* ptr, std::size_t size) noexcept`. If this function with `size` parameter is defined, the program shall also define the version without the `size` parameter. [*Note:* The default behavior below may change in the future, which will require replacing both deallocation functions when replacing the allocation function. — *end note*]

2

Requires: *ptr* shall be a null pointer or its value shall be the value returned by an earlier call to `operator new[](std::size_t)` or `operator new[](std::size_t,const std::nothrow_t&)` which has not been invalidated by an intervening call to `operator delete[](void*)`.

1 *Requires:* If present, the `std::size_t size` argument must equal the size argument passed to the allocation function that returned `ptr`.

2 *Required behavior:* Calls to `operator delete[](void* ptr, std::size_t size)` may be changed to calls to `operator delete[](void* ptr)` without affecting memory allocation. [*Note:* A conforming implementation is for `operator delete[](void* ptr, std::size_t size)` to simply call `operator delete[](void* ptr)`. — *end note*]

3 *Requires:* If an implementation has strict pointer safety (3.2532.0.1) then `ptr` shall be a safely-derived pointer.

4

Default behavior: `operator delete[](void* ptr, std::size_t size)` calls `operator delete[](ptr)`, and `operator delete[](void* ptr)` calls `operator delete(ptr)`.

```
void operator delete[](void* ptr, const std::nothrow_t&) noexcept;
void operator delete[](void* ptr, std::size_t size, const std::nothrow_t&) noexcept;
```

1 *Effects:* The *deallocation function* (3.2428.0.1) called by the implementation to render the value of `ptr` invalid when the constructor invoked from a nothrow placement version of the array *new-expression* throws an exception.

2 *Replaceable:* a C++ program may define a function with signature `void operator delete[](void* ptr, const std::nothrow_t&) noexcept` that displaces the default version defined by the C++ standard library. If this function (without `size` parameter) is defined, the program should also define `void operator delete[](void* ptr, std::size_t size, const std::nothrow_t&) noexcept`. If this function with `size` parameter is defined, the program shall also define the version without the `size` parameter. [*Note:* The default behavior below may change in the future, which will require replacing both deallocation functions when replacing the allocation function. — *end note*]

3 *Requires:* If an implementation has strict pointer safety (3.2532.0.1) then `ptr` shall be a safely-derived pointer.

4 *Requires:* If present, the `std::size_t size` argument must equal the size argument passed to the allocation function that returned `ptr`.

5 *Required behavior:* Calls to `operator delete[](void* ptr, std::size_t size, const std::nothrow_t&)` may be changed to calls to `operator delete[](void* ptr, const std::nothrow_t&)` without affecting memory allocation. [*Note:* A conforming implementation is for `operator delete[](void* ptr, std::size_t size, const std::nothrow_t&)` to simply call `operator delete[](void* ptr, const std::nothrow_t&)`. — *end note*]

6 *Default behavior:* `operator delete[](void* ptr, std::size_t size, const std::nothrow_t&)` calls `operator delete[](ptr, std::nothrow)`, and `operator delete[](void* ptr, const std::nothrow_t&)` calls `operator delete[](ptr)`.

18.203.0.1 Placement forms [new.delete.placement]

1

These functions are reserved, a C++ program may not define functions that displace the versions in the Standard C++ library (17.194.1). The provisions of (3.2331.1) do not apply to these reserved placement forms of `operator new` and `operator delete`.

`void* operator new(std::size_t size, void* ptr) noexcept;`

1 *Returns:* `ptr`.

2

Remarks: Intentionally performs no other action.

1 [*Example:* This can be useful for constructing an object at a known address:

```
void* place = operator new(sizeof(Something));
Something* p = new (place) Something();
```

— *end example*]

`void* operator new[](std::size_t size, void* ptr) noexcept;`

2 *Returns:* `ptr`.

3 *Remarks:* Intentionally performs no other action.

`void operator delete(void* ptr, void*) noexcept;`

4 *Effects:* Intentionally performs no action.

5 *Requires:* If an implementation has strict pointer safety (3.2532.0.1) then `ptr` shall be a safely-derived pointer.

6 *Remarks:* Default function called when any part of the initialization in a placement new expression that invokes the library's non-array placement operator new terminates by throwing an exception (5.2221.1).

`void operator delete[](void* ptr, void*) noexcept;`

7 *Effects:* Intentionally performs no action.

8 *Requires:* If an implementation has strict pointer safety (3.2532.0.1) then `ptr` shall be a safely-derived pointer.

9 *Remarks:* Default function called when any part of the initialization in a placement new expression that invokes the library's array placement operator new terminates by throwing an exception (5.2221.1).

18.220.0.1 Data races [new.delete.dataraces]

1

For purposes of determining the existence of data races, the library versions of `operator new`, user replacement versions of global `operator new`, the C standard library functions `calloc` and `malloc`, the library versions of `operator delete`, user replacement versions of `operator delete`, the C standard library function `free`, and the C standard library function `realloc` shall not introduce a data race (17.607.0.1). Calls to these functions that allocate or deallocate a particular unit of storage shall occur in a single total order, and each such deallocation call shall happen before (1.859) the next allocation (if any) in this order.

18.223.1 Storage allocation errors [alloc.errors]

18.223.1.1 Class bad_alloc [bad.alloc]

```
namespace std {
  class bad_alloc : public exception {
  public:
    bad_alloc() noexcept;
    bad_alloc(const bad_alloc&) noexcept;
    bad_alloc& operator=(const bad_alloc&) noexcept;
    virtual const char* what() const noexcept;
  };
}
```

1 The class `bad_alloc` defines the type of objects thrown as exceptions by the implementation to report a failure to allocate storage.

`bad_alloc() noexcept;`

2 *Effects:* Constructs an object of class `bad_alloc`.

3 *Remarks:* The result of calling `what()` on the newly constructed object is implementation-defined.

```
bad_alloc(const bad_alloc&) noexcept;
bad_alloc& operator=(const bad_alloc&) noexcept;
```

1 *Effects:* Copies an object of class `bad_alloc`.

`virtual const char* what() const noexcept;`

1 *Returns:* An implementation-defined NTBS.

18.225.0.1 Class bad_array_new_length [new.badlength]

```
namespace std {
  class bad_array_new_length : public bad_alloc {
  public:
    bad_array_new_length() noexcept;
  };
}
```

1 The class `bad_array_new_length` defines the type of objects thrown as exceptions by the implementation to report an attempt to allocate an array of size less than zero or greater than an implementation-defined limit (5.2221.1).

`bad_array_new_length() noexcept;`

1 *Effects:* constructs an object of class `bad_array_new_length`.

2 *Remarks:* the result of calling `what()` on the newly constructed object is implementation-defined.

18.226.0.1 Type new_handler [new.handler]

```
typedef void (*new_handler)();
```

1 The type of a *handler function* to be called by `operator new()` or `operator new[]()` (18.159.1) when they cannot satisfy a request for additional storage.

2 *Required behavior:* A `new_handler` shall perform one of the following:

(2.1) — make more storage available for allocation and then return;

(2.2) — throw an exception of type `bad_alloc` or a class derived from `bad_alloc`;

(2.3) — terminate execution of the program without returning to the caller;

18.226.0.2 `set_new_handler` [set.new.handler]

```
new_handler set_new_handler(new_handler new_p) noexcept;
```

1 *Effects:* Establishes the function designated by `new_p` as the current `new_handler`.

2 *Returns:* The previous `new_handler`.

3 *Remarks:* The initial `new_handler` is a null pointer.

18.226.0.3 `get_new_handler` [get.new.handler]

```
new_handler get_new_handler() noexcept;
```

1 *Returns:* The current `new_handler`. [*Note:* This may be a null pointer value. — *end note*]

18.227 Type identification [support.rtti]

1 The header `<typeinfo>` defines a type associated with type information generated by the implementation. It also defines two types for reporting dynamic type identification errors.

Header `<typeinfo>` synopsis

```
namespace std {
  class type_info;
  class bad_cast;
  class bad_typeid;
}
```

See also: 5.1295.1, 5.1393.1.

18.227.1 Class `type_info` [type.info]

```
namespace std {
  class type_info {
  public:
    virtual ~type_info();
    bool operator==(const type_info& rhs) const noexcept;
    bool operator!=(const type_info& rhs) const noexcept;
    bool before(const type_info& rhs) const noexcept;
    size_t hash_code() const noexcept;
    const char* name() const noexcept;

    type_info(const type_info& rhs) = delete;            // cannot be copied
    type_info& operator=(const type_info& rhs) = delete; // cannot be copied
  };
}
```

1

The class `type_info` describes type information generated by the implementation. Objects of this class effectively store a pointer to a name for the type, and an encoded value suitable for comparing two types for

equality or collating order. The names, encoding rule, and collating sequence for types are all unspecified and may differ between programs.

```
bool operator==(const type_info& rhs) const noexcept;
```

1 *Effects:* Compares the current object with `rhs`.

1 *Returns:* `true` if the two values describe the same type.

```
bool operator!=(const type_info& rhs) const noexcept;
```

1 *Returns:* `!(*this == rhs)`.

```
bool before(const type_info& rhs) const noexcept;
```

2 *Effects:* Compares the current object with `rhs`.

3 *Returns:* `true` if `*this` precedes `rhs` in the implementation's collation order.

```
size_t hash_code() const noexcept;
```

4 *Returns:* An unspecified value, except that within a single execution of the program, it shall return the same value for any two `type_info` objects which compare equal.

1 *Remark:* an implementation should return different values for two `type_info` objects which do not compare equal.

```
const char* name() const noexcept;
```

1 *Returns:* An implementation-defined NTBS.

2 *Remarks:* The message may be a null-terminated multibyte string (17.26.0.0.0.1), suitable for conversion and display as a `wstring` (21.26, 22.34.0.1)

18.240.1 Class `bad_cast` [bad.cast]

```
namespace std {
  class bad_cast : public exception {
  public:
    bad_cast() noexcept;
    bad_cast(const bad_cast&) noexcept;
    bad_cast& operator=(const bad_cast&) noexcept;
    virtual const char* what() const noexcept;
  };
}
```

1 The class `bad_cast` defines the type of objects thrown as exceptions by the implementation to report the execution of an invalid *dynamic-cast* expression (5.1295.1).

```
bad_cast() noexcept;
```

2 *Effects:* Constructs an object of class `bad_cast`.

3 *Remarks:* The result of calling `what()` on the newly constructed object is implementation-defined.

```
bad_cast(const bad_cast&) noexcept;
bad_cast& operator=(const bad_cast&) noexcept;
```

4 *Effects:* Copies an object of class `bad_cast`.

```
virtual const char* what() const noexcept;
```

5 *Returns:* An implementation-defined NTBS.

6 *Remarks:* The message may be a null-terminated multibyte string (17.26.0.0.0.1), suitable for conversion and display as a `wstring` (21.26, 22.34.0.1)

18.240.2 Class `bad_typeid` [bad.typeid]

```
namespace std {
  class bad_typeid : public exception {
  public:
    bad_typeid() noexcept;
    bad_typeid(const bad_typeid&) noexcept;
    bad_typeid& operator=(const bad_typeid&) noexcept;
    virtual const char* what() const noexcept;
  };
}
```

1 The class `bad_typeid` defines the type of objects thrown as exceptions by the implementation to report a null pointer in a *typeid* expression (5.1393.1).

```
bad_typeid() noexcept;
```

2 *Effects:* Constructs an object of class `bad_typeid`.

3 *Remarks:* The result of calling `what()` on the newly constructed object is implementation-defined.

```
bad_typeid(const bad_typeid&) noexcept;
bad_typeid& operator=(const bad_typeid&) noexcept;
```

4 *Effects:* Copies an object of class `bad_typeid`.

```
virtual const char* what() const noexcept;
```

5 *Returns:* An implementation-defined NTBS.

6 *Remarks:* The message may be a null-terminated multibyte string (17.26.0.0.0.1), suitable for conversion and display as a `wstring` (21.26, 22.34.0.1)

18.241 Exception handling [support.exception]

1 The header `<exception>` defines several types and functions related to the handling of exceptions in a C++ program.

Header `<exception>` synopsis

```
namespace std {
  class exception;
  class bad_exception;
  class nested_exception;

  typedef void (*unexpected_handler)();
  unexpected_handler get_unexpected() noexcept;
  unexpected_handler set_unexpected(unexpected_handler f) noexcept;
  [[noreturn]] void unexpected();
```

```
  typedef void (*terminate_handler)();
  terminate_handler get_terminate() noexcept;
  terminate_handler set_terminate(terminate_handler f) noexcept;
  [[noreturn]] void terminate() noexcept;

  bool uncaught_exception() noexcept;

  typedef unspecified  exception_ptr;

  exception_ptr current_exception() noexcept;
  [[noreturn]] void rethrow_exception(exception_ptr p);
  template<class E> exception_ptr make_exception_ptr(E e) noexcept;

  template <class T> [[noreturn]] void throw_with_nested(T&& t);
  template <class E> void rethrow_if_nested(const E& e);
}
```

See also: 15.308.

18.241.1 Class `exception` [exception]

```
namespace std {
  class exception {
  public:
    exception() noexcept;
    exception(const exception&) noexcept;
    exception& operator=(const exception&) noexcept;
    virtual ~exception();
    virtual const char* what() const noexcept;
  };
}
```

1 The class `exception` defines the base class for the types of objects thrown as exceptions by C++ standard library components, and certain expressions, to report errors detected during program execution.

2 Each standard library class `T` that derives from class `exception` shall have a publicly accessible copy constructor and a publicly accessible copy assignment operator that do not exit with an exception. These member functions shall meet the following postcondition: If two objects `lhs` and `rhs` both have dynamic type `T` and `lhs` is a copy of `rhs`, then `strcmp(lhs.what(), rhs.what())` shall equal 0.

`exception() noexcept;`

1 *Effects:* Constructs an object of class `exception`.

```
exception(const exception& rhs) noexcept;
exception& operator=(const exception& rhs) noexcept;
```

2 *Effects:* Copies an `exception` object.

3 *Postcondition:* If `*this` and `rhs` both have dynamic type `exception` then `strcmp(what(), rhs.what())` shall equal 0.

`virtual ~exception();`

4 *Effects:* Destroys an object of class `exception`.

`virtual const char* what() const noexcept;`

5 *Returns:* An implementation-defined NTBS.

6 *Remarks:* The message may be a null-terminated multibyte string (17.26.0.0.0.1), suitable for conversion and display as a `wstring` (21.26, 22.34.0.1). The return value remains valid until the exception object from which it is obtained is destroyed or a non-`const` member function of the exception object is called.

18.247.1 Class `bad_exception` [bad.exception]

```
namespace std {
  class bad_exception : public exception {
  public:
    bad_exception() noexcept;
    bad_exception(const bad_exception&) noexcept;
    bad_exception& operator=(const bad_exception&) noexcept;
    virtual const char* what() const noexcept;
  };
}
```

1 The class `bad_exception` defines the type of objects thrown as described in (15.370.1).

`bad_exception() noexcept;`

2 *Effects:* Constructs an object of class `bad_exception`.

3 *Remarks:* The result of calling `what()` on the newly constructed object is implementation-defined.

```
bad_exception(const bad_exception&) noexcept;
bad_exception& operator=(const bad_exception&) noexcept;
```

4 *Effects:* Copies an object of class `bad_exception`.

`virtual const char* what() const noexcept;`

5 *Returns:* An implementation-defined NTBS.

6 *Remarks:* The message may be a null-terminated multibyte string (17.26.0.0.0.1), suitable for conversion and display as a `wstring` (21.26, 22.34.0.1).

18.247.2 Abnormal termination [exception.terminate]

18.247.2.1 Type `terminate_handler` [terminate.handler]

`typedef void (*terminate_handler)();`

1 The type of a *handler function* to be called by `std::terminate()` when terminating exception processing.

2 *Required behavior:* A `terminate_handler` shall terminate execution of the program without returning to the caller.

1 *Default behavior:* The implementation's default `terminate_handler` calls `abort()`.

18.250.0.1 `set_terminate` [set.terminate]

```
terminate_handler set_terminate(terminate_handler f) noexcept;
```

1 *Effects:* Establishes the function designated by `f` as the current handler function for terminating exception processing.

1 *Remarks:* It is unspecified whether a null pointer value designates the default `terminate_handler`.

2 *Returns:* The previous `terminate_handler`.

18.251.0.1 `get_terminate` [get.terminate]

```
terminate_handler get_terminate() noexcept;
```

1 *Returns:* The current `terminate_handler`. [*Note:* This may be a null pointer value. — *end note*]

18.251.0.2 `terminate` [terminate]

```
[[noreturn]] void terminate() noexcept;
```

1 *Remarks:* Called by the implementation when exception handling must be abandoned for any of several reasons (15.308.1), in effect immediately after throwing the exception. May also be called directly by the program.

1 *Effects:* Calls the current `terminate_handler` function. [*Note:* A default `terminate_handler` is always considered a callable handler in this context. — *end note*]

18.253.1 `uncaught_exception` [uncaught]

```
bool uncaught_exception() noexcept;
```

1 *Returns:* `true` after the current thread has initialized an exception object (15.7) until a handler for the exception (including `std::unexpected()` or `std::terminate()`) is activated (15.146). [*Note:* This includes stack unwinding (15.89). — *end note*]

2 *Remarks:* When `uncaught_exception()` returns `true`, throwing an exception can result in a call of `std::terminate()` (15.308.1).

18.254.1 Exception propagation [propagation]

```
typedef unspecified  exception_ptr;
```

1 The type exception_ptr can be used to refer to an exception object.

1 `exception_ptr` shall satisfy the requirements of `NullablePointer` (17.153.0.1).

1 Two non-null values of type `exception_ptr` are equivalent and compare equal if and only if they refer to the same exception.

1 The default constructor of `exception_ptr` produces the null value of the type.

1

`exception_ptr` shall not be implicitly convertible to any arithmetic, enumeration, or pointer type.

1

[*Note:* An implementation might use a reference-counted smart pointer as `exception_ptr`. — *end note*]

1

For purposes of determining the presence of a data race, operations on `exception_ptr` objects shall access and modify only the `exception_ptr` objects themselves and not the exceptions they refer to. Use of `rethrow_exception` on `exception_ptr` objects that refer to the same exception object shall not introduce a data race. [*Note:* if `rethrow_exception` rethrows the same exception object (rather than a copy), concurrent access to that rethrown exception object may introduce a data race. Changes in the number of `exception_ptr` objects that refer to a particular exception do not introduce a data race. — *end note*]

```
exception_ptr current_exception() noexcept;
```

1

Returns: An `exception_ptr` object that refers to the currently handled exception (15.146) or a copy of the currently handled exception, or a null `exception_ptr` object if no exception is being handled. The referenced object shall remain valid at least as long as there is an `exception_ptr` object that refers to it. If the function needs to allocate memory and the attempt fails, it returns an `exception_ptr` object that refers to an instance of `bad_alloc`. It is unspecified whether the return values of two successive calls to `current_exception` refer to the same exception object. [*Note:* That is, it is unspecified whether `current_exception` creates a new copy each time it is called. — *end note*] If the attempt to copy the current exception object throws an exception, the function returns an `exception_ptr` object that refers to the thrown exception or, if this is not possible, to an instance of `bad_exception`. [*Note:* The copy constructor of the thrown exception may also fail, so the implementation is allowed to substitute a `bad_exception` object to avoid infinite recursion. — *end note*]

```
[[noreturn]] void rethrow_exception(exception_ptr p);
```

1 *Requires:* `p` shall not be a null pointer.

2 *Throws:* the exception object to which `p` refers.

```
template<class E> exception_ptr make_exception_ptr(E e) noexcept;
```

3 *Effects:* Creates an `exception_ptr` object that refers to a copy of `e`, as if

```
try {
  throw e;
} catch(...) {
  return current_exception();
}
```

4 [*Note:* This function is provided for convenience and efficiency reasons. — *end note*]

18.265.1 `nested_exception` [except.nested]

```
namespace std {
  class nested_exception {
  public:
    nested_exception() noexcept;
    nested_exception(const nested_exception&) noexcept = default;
```

```
    nested_exception& operator=(const nested_exception&) noexcept = default;
    virtual ~nested_exception() = default;

    // access functions
    [[noreturn]] void rethrow_nested() const;
    exception_ptr nested_ptr() const noexcept;
  };

  template<class T> [[noreturn]] void throw_with_nested(T&& t);
  template <class E> void rethrow_if_nested(const E& e);
}
```

1 The class `nested_exception` is designed for use as a mixin through multiple inheritance. It captures the currently handled exception and stores it for later use.

2 [*Note:* `nested_exception` has a virtual destructor to make it a polymorphic class. Its presence can be tested for with `dynamic_cast`. — *end note*]

`nested_exception() noexcept;`

1 *Effects:* The constructor calls `current_exception()` and stores the returned value.

`[[noreturn]] void rethrow_nested() const;`

2 *Effects:* If `nested_ptr()` returns a null pointer, the function calls `std::terminate()`. Otherwise, it throws the stored exception captured by `*this`.

`exception_ptr nested_ptr() const noexcept;`

3 *Returns:* The stored exception captured by this `nested_exception` object.

`template <class T> [[noreturn]] void throw_with_nested(T&& t);`

1 Let `U` be `remove_reference_t<T>`.

2 *Requires:* `U` shall be `CopyConstructible`.

3 *Throws:* if `U` is a non-union class type not derived from `nested_exception`, an exception of unspecified type that is publicly derived from both `U` and `nested_exception` and constructed from `std::forward<T>(t)`, otherwise `std::forward<T>(t)`.

`template <class E> void rethrow_if_nested(const E& e);`

1 *Effects:* If the dynamic type of `e` is publicly and unambiguously derived from `nested_exception`, calls `dynamic_cast<const nested_exception&>(e).rethrow_nested()`.

18.274 Initializer lists [support.initlist]

1 The header `<initializer_list>` defines a class template and several support functions related to list-initialization (see 8.2435.1).

Header `<initializer_list>` synopsis

```
namespace std {
  template<class E> class initializer_list {
  public:
```

```
    typedef E value_type;
    typedef const E& reference;
    typedef const E& const_reference;
    typedef size_t size_type;

    typedef const E* iterator;
    typedef const E* const_iterator;

    constexpr initializer_list() noexcept;

    constexpr size_t size() const noexcept;       // number of elements
    constexpr const E* begin() const noexcept;    // first element
    constexpr const E* end() const noexcept;      // one past the last element
  };

  // 18.294.3 initializer list range access
  template<class E> constexpr const E* begin(initializer_list<E> il) noexcept;
  template<class E> constexpr const E* end(initializer_list<E> il) noexcept;
}
```

2 An object of type `initializer_list<E>` provides access to an array of objects of type `const E`. [*Note:* A pair of pointers or a pointer plus a length would be obvious representations for `initializer_list`. `initializer_list` is used to implement initializer lists as specified in 8.2435.1. Copying an initializer list does not copy the underlying elements. — *end note*]

18.294.1 Initializer list constructors [support.initlist.cons]

`constexpr initializer_list() noexcept;`

1 *Effects:* constructs an empty `initializer_list` object.

2 *Postcondition:* `size() == 0`

18.294.2 Initializer list access [support.initlist.access]

`constexpr const E* begin() const noexcept;`

1 *Returns:* A pointer to the beginning of the array. If `size() == 0` the values of `begin()` and `end()` are unspecified but they shall be identical.

`constexpr const E* end() const noexcept;`

2 *Returns:* `begin() + size()`

`constexpr size_t size() const noexcept;`

3 *Returns:* The number of elements in the array.

4 *Complexity:* Constant time.

18.294.3 Initializer list range access [support.initlist.range]

`template<class E> constexpr const E* begin(initializer_list<E> il) noexcept;`

1 *Returns:* `il.begin()`.

`template<class E> constexpr const E* end(initializer_list<E> il) noexcept;`

2 *Returns:* `il.end()`.

18.295 Other runtime support [support.runtime]

1

Headers `<csetjmp>` (nonlocal jumps), `<csignal>` (signal handling), `<cstdalign> (alignment)`, `<cstdarg>` (variable arguments), `<cstdbool>` (`__bool_true_false_are_defined`). `<cstdlib>` (runtime environment `getenv()`, `system()`), and `<ctime>` (system clock `clock()`, `time()`) provide further compatibility with C code.

1

The contents of these headers are the same as the Standard C library headers `<setjmp.h>`, `<signal.h>`, `<stdalign.h>`, `<stdarg.h>`, `<stdbool.h>`, `<stdlib.h>`, and `<time.h>`, respectively, with the following changes:

1

The restrictions that ISO C places on the second parameter to the `va_start()` macro in header `<stdarg.h>` are different in this International Standard. The parameter `parmN` is the identifier of the rightmost parameter in the variable parameter list of the function definition (the one just before the `...`).[227] If the parameter `parmN` is of a reference type, or of a type that is not compatible with the type that results when passing an argument for which there is no parameter, the behavior is undefined.

See also: ISO C 4.8.1.1.

1

The function signature `longjmp(jmp_buf jbuf, int val)` has more restricted behavior in this International Standard. A `setjmp`/`longjmp` call pair has undefined behavior if replacing the `setjmp` and `longjmp` by `catch` and `throw` would invoke any non-trivial destructors for any automatic objects.

See also: ISO C 7.10.4, 7.8, 7.6, 7.12.

1

Calls to the function `getenv` shall not introduce a data race (17.607.0.1) provided that nothing modifies the environment. [*Note:* Calls to the POSIX functions `setenv` and `putenv` modify the environment. — *end note*]

1

A call to the `setlocale` function may introduce a data race with other calls to the `setlocale` function or with calls to functions that are affected by the current C locale. The implementation shall behave as if no library function other than `locale::global()` calls the `setlocale` function.

1

The header `<cstdalign>` and the header `<stdalign.h>` shall not define a macro named `alignas`.

1

The header `<cstdbool>` and the header `<stdbool.h>` shall not define macros named `bool`, `true`, or `false`.

1 A call to the function `signal` synchronizes with any resulting invocation of the signal handler so installed.

2

The common subset of the C and C++ languages consists of all declarations, definitions, and expressions that may appear in a well formed C++ program and also in a conforming C program. A POF ("plain old function") is a function that uses only features from this common subset, and that does not directly or indirectly use any function that is not a POF, except that it may use plain lock-free atomic operations. A *plain lock-free atomic operation* is an invocation of a function f from Clause 29, such that f is not a member function, and either f is the function `atomic_is_lock_free`, or for every atomic argument `A` passed to

227) Note that `va_start` is required to work as specified even if unary `operator&` is overloaded for the type of `parmN`.

f, `atomic_is_lock_free(A)` yields `true`. All signal handlers shall have C linkage. The behavior of any function other than a POF used as a signal handler in a C++ program is implementation-defined.[228]

Table 34 — Header `<csetjmp>` synopsis

Type	Name(s)
Macro:	`setjmp`
Type:	`jmp_buf`
Function:	`longjmp`

Table 35 — Header `<csignal>` synopsis

Type	Name(s)			
Macros:	`SIGABRT`	`SIGILL`	`SIGSEGV`	`SIG_DFL`
	`SIG_IGN`	`SIGFPE`	`SIGINT`	`SIGTERM`	`SIG_ERR`
Type:	`sig_atomic_t`			
Functions:	`raise`	`signal`		

Table 36 — Header `<cstdalign>` synopsis

Type	Name(s)
Macro:	`__alignas_is_defined`

Table 37 — Header `<cstdarg>` synopsis

Type	Name(s)		
Macros:	`va_arg`	`va_end`	`va_start`
	`va_copy`		
Type:	`va_list`		

Table 38 — Header `<cstdbool>` synopsis

Type	Name(s)
Macro:	`__bool_true_false_are_defined`

Table 39 — Header `<cstdlib>` synopsis

Type	Name(s)	
Functions:	`getenv`	`system`

228) In particular, a signal handler using exception handling is very likely to have problems. Also, invoking `std::exit` may cause destruction of objects, including those of the standard library implementation, which, in general, yields undefined behavior in a signal handler (see 1.430).

Table 40 — Header `<ctime>` synopsis

Type	Name(s)
Macro:	`CLOCKS_PER_SEC`
Type:	`clock_t`
Function:	`clock`

19 Diagnostics library [diagnostics]

19.1 General [diagnostics.general]

1

This Clause describes components that C++ programs may use to detect and report error conditions.

1 The following subclauses describe components for reporting several kinds of exceptional conditions, documenting program assertions, and a global variable for error number codes, as summarized in Table 41.

Table 41 — Diagnostics library summary

Subclause		Header(s)
19.3	Exception classes	`<stdexcept>`
19.8	Assertions	`<cassert>`
19.12	Error numbers	`<cerrno>`
19.14	System error support	`<system_error>`

19.3 Exception classes [std.exceptions]

1 The Standard C++ library provides classes to be used to report certain errors (17.677.0.1) in C++ programs. In the error model reflected in these classes, errors are divided into two broad categories: *logic* errors and *runtime* errors.

2 The distinguishing characteristic of logic errors is that they are due to errors in the internal logic of the program. In theory, they are preventable.

3 By contrast, runtime errors are due to events beyond the scope of the program. They cannot be easily predicted in advance. The header `<stdexcept>` defines several types of predefined exceptions for reporting errors in a C++ program. These exceptions are related by inheritance.

Header `<stdexcept>` synopsis

```
namespace std {
  class logic_error;
    class domain_error;
    class invalid_argument;
    class length_error;
    class out_of_range;
  class runtime_error;
    class range_error;
    class overflow_error;
    class underflow_error;
}
```

19.3.1 Class `logic_error` [logic.error]

```
namespace std {
  class logic_error : public exception {
  public:
    explicit logic_error(const string& what_arg);
    explicit logic_error(const char* what_arg);
```

```
  };
}
```

1 The class `logic_error` defines the type of objects thrown as exceptions to report errors presumably detectable before the program executes, such as violations of logical preconditions or class invariants.

`logic_error(const string& what_arg);`

2 *Effects:* Constructs an object of class `logic_error`.

3 *Postcondition:* `strcmp(what(), what_arg.c_str()) == 0`.

`logic_error(const char* what_arg);`

4 *Effects:* Constructs an object of class `logic_error`.

5 *Postcondition:* `strcmp(what(), what_arg) == 0`.

19.3.2 Class `domain_error` [domain.error]

```
namespace std {
  class domain_error : public logic_error {
  public:
    explicit domain_error(const string& what_arg);
    explicit domain_error(const char* what_arg);
  };
}
```

1 The class `domain_error` defines the type of objects thrown as exceptions by the implementation to report domain errors.

`domain_error(const string& what_arg);`

2 *Effects:* Constructs an object of class `domain_error`.

3 *Postcondition:* `strcmp(what(), what_arg.c_str()) == 0`.

`domain_error(const char* what_arg);`

4 *Effects:* Constructs an object of class `domain_error`.

5 *Postcondition:* `strcmp(what(), what_arg) == 0`.

19.3.3 Class `invalid_argument` [invalid.argument]

```
namespace std {
  class invalid_argument : public logic_error {
  public:
    explicit invalid_argument(const string& what_arg);
    explicit invalid_argument(const char* what_arg);
  };
}
```

1 The class `invalid_argument` defines the type of objects thrown as exceptions to report an invalid argument.

`invalid_argument(const string& what_arg);`

2 *Effects:* Constructs an object of class `invalid_argument`.

3 *Postcondition:* `strcmp(what(), what_arg.c_str()) == 0`.

`invalid_argument(const char* what_arg);`

4 *Effects:* Constructs an object of class `invalid_argument`.

5 *Postcondition:* `strcmp(what(), what_arg) == 0`.

19.3.4 Class `length_error` [length.error]

```
namespace std {
  class length_error : public logic_error {
  public:
    explicit length_error(const string& what_arg);
    explicit length_error(const char* what_arg);
  };
}
```

1

The class `length_error` defines the type of objects thrown as exceptions to report an attempt to produce an object whose length exceeds its maximum allowable size.

`length_error(const string& what_arg);`

1 *Effects:* Constructs an object of class `length_error`.

2 *Postcondition:* `strcmp(what(), what_arg.c_str()) == 0`.

`length_error(const char* what_arg);`

3 *Effects:* Constructs an object of class `length_error`.

4 *Postcondition:* `strcmp(what(), what_arg) == 0`.

19.4.1 Class `out_of_range` [out.of.range]

```
namespace std {
  class out_of_range : public logic_error {
  public:
    explicit out_of_range(const string& what_arg);
    explicit out_of_range(const char* what_arg);
  };
}
```

1

The class `out_of_range` defines the type of objects thrown as exceptions to report an argument value not in its expected range.

`out_of_range(const string& what_arg);`

1 *Effects:* Constructs an object of class `out_of_range`.

2 *Postcondition:* `strcmp(what(), what_arg.c_str()) == 0`.

`out_of_range(const char* what_arg);`

3 *Effects:* Constructs an object of class `out_of_range`.

4 *Postcondition:* `strcmp(what(), what_arg) == 0`.

19.5.1 Class `runtime_error` [runtime.error]

```
namespace std {
  class runtime_error : public exception {
  public:
    explicit runtime_error(const string& what_arg);
    explicit runtime_error(const char* what_arg);
  };
}
```

1 The class `runtime_error` defines the type of objects thrown as exceptions to report errors presumably detectable only when the program executes.

`runtime_error(const string& what_arg);`

2

Effects: Constructs an object of class `runtime_error`.

1 *Postcondition:* `strcmp(what(), what_arg.c_str()) == 0`.

`runtime_error(const char* what_arg);`

2 *Effects:* Constructs an object of class `runtime_error`.

3 *Postcondition:* `strcmp(what(), what_arg) == 0`.

19.6.1 Class `range_error` [range.error]

```
namespace std {
  class range_error : public runtime_error {
  public:
    explicit range_error(const string& what_arg);
    explicit range_error(const char* what_arg);
  };
}
```

1

The class `range_error` defines the type of objects thrown as exceptions to report range errors in internal computations.

`range_error(const string& what_arg);`

1 *Effects:* Constructs an object of class `range_error`.

2 *Postcondition:* `strcmp(what(), what_arg.c_str()) == 0`.

`range_error(const char* what_arg);`

3 *Effects:* Constructs an object of class `range_error`.

4 *Postcondition:* `strcmp(what(), what_arg) == 0`.

19.7.1 Class `overflow_error` [overflow.error]

```
namespace std {
  class overflow_error : public runtime_error {
  public:
    explicit overflow_error(const string& what_arg);
    explicit overflow_error(const char* what_arg);
  };
}
```

1 The class `overflow_error` defines the type of objects thrown as exceptions to report an arithmetic overflow error.

`overflow_error(const string& what_arg);`

2 *Effects:* Constructs an object of class `overflow_error`.

3 *Postcondition:* `strcmp(what(), what_arg.c_str()) == 0`.

`overflow_error(const char* what_arg);`

4 *Effects:* Constructs an object of class `overflow_error`.

5 *Postcondition:* `strcmp(what(), what_arg) == 0`.

19.7.2 Class `underflow_error` [underflow.error]

```
namespace std {
  class underflow_error : public runtime_error {
  public:
    explicit underflow_error(const string& what_arg);
    explicit underflow_error(const char* what_arg);
  };
}
```

1 The class `underflow_error` defines the type of objects thrown as exceptions to report an arithmetic underflow error.

`underflow_error(const string& what_arg);`

2 *Effects:* Constructs an object of class `underflow_error`.

3 *Postcondition:* `strcmp(what(), what_arg.c_str()) == 0`.

`underflow_error(const char* what_arg);`

4 *Effects:* Constructs an object of class `underflow_error`.

5 *Postcondition:* `strcmp(what(), what_arg) == 0`.

19.8 Assertions [assertions]

1

The header `<cassert>`, described in (Table 42), provides a macro for documenting C++ program assertions and a mechanism for disabling the assertion checks.

Table 42 — Header `<cassert>` synopsis

Type	Name(s)
Macro:	`assert`

1

The contents are the same as the Standard C library header `<assert.h>`.

See also: ISO C 7.2.

19.12 Error numbers [errno]

1

The header `<cerrno>` is described in Table 43. Its contents are the same as the POSIX header `<errno.h>`, except that `errno` shall be defined as a macro. [*Note:* The intent is to remain in close alignment with the POSIX standard. — *end note*] A separate `errno` value shall be provided for each thread.

19.14 System error support [syserr]

1

This subclause describes components that the standard library and C++ programs may use to report error conditions originating from the operating system or other low-level application program interfaces.

Table 43 — Header `<cerrno>` synopsis

Type	Name(s)				
Macros:	ECONNREFUSED	EIO	ENODEV	ENOTEMPTY	ERANGE
E2BIG	ECONNRESET	EISCONN	ENOENT	ENOTRECOVERABLE	EROFS
EACCES	EDEADLK	EISDIR	ENOEXEC	ENOTSOCK	ESPIPE
EADDRINUSE	EDESTADDRREQ	ELOOP	ENOLCK	ENOTSUP	ESRCH
EADDRNOTAVAIL	EDOM	EMFILE	ENOLINK	ENOTTY	ETIME
EAFNOSUPPORT	EEXIST	EMLINK	ENOMEM	ENXIO	ETIMEDOUT
EAGAIN	EFAULT	EMSGSIZE	ENOMSG	EOPNOTSUPP	ETXTBSY
EALREADY	EFBIG	ENAMETOOLONG	ENOPROTOOPT	EOVERFLOW	EWOULDBLOCK
EBADF	EHOSTUNREACH	ENETDOWN	ENOSPC	EOWNERDEAD	EXDEV
EBADMSG	EIDRM	ENETRESET	ENOSR	EPERM	errno
EBUSY	EILSEQ	ENETUNREACH	ENOSTR	EPIPE	
ECANCELED	EINPROGRESS	ENFILE	ENOSYS	EPROTO	
ECHILD	EINTR	ENOBUFS	ENOTCONN	EPROTONOSUPPORT	
ECONNABORTED	EINVAL	ENODATA	ENOTDIR	EPROTOTYPE	

1 Components described in this subclause shall not change the value of `errno` (19.12). Implementations should leave the error states provided by other libraries unchanged.

Header `<system_error>` synopsis

```
namespace std {
  class error_category;
  const error_category& generic_category() noexcept;
  const error_category& system_category() noexcept;

  class error_code;
  class error_condition;
  class system_error;

  template <class T>
  struct is_error_code_enum : public false_type {};

  template <class T>
  struct is_error_condition_enum : public false_type {};

  enum class errc {
    address_family_not_supported,       // EAFNOSUPPORT
    address_in_use,                     // EADDRINUSE
    address_not_available,              // EADDRNOTAVAIL
    already_connected,                  // EISCONN
    argument_list_too_long,             // E2BIG
    argument_out_of_domain,             // EDOM
    bad_address,                        // EFAULT
    bad_file_descriptor,                // EBADF
    bad_message,                        // EBADMSG
    broken_pipe,                        // EPIPE
    connection_aborted,                 // ECONNABORTED
    connection_already_in_progress,     // EALREADY
    connection_refused,                 // ECONNREFUSED
    connection_reset,                   // ECONNRESET
    cross_device_link,                  // EXDEV
```

```
destination_address_required,       // EDESTADDRREQ
device_or_resource_busy,            // EBUSY
directory_not_empty,                // ENOTEMPTY
executable_format_error,            // ENOEXEC
file_exists,                        // EEXIST
file_too_large,                     // EFBIG
filename_too_long,                  // ENAMETOOLONG
function_not_supported,             // ENOSYS
host_unreachable,                   // EHOSTUNREACH
identifier_removed,                 // EIDRM
illegal_byte_sequence,              // EILSEQ
inappropriate_io_control_operation, // ENOTTY
interrupted,                        // EINTR
invalid_argument,                   // EINVAL
invalid_seek,                       // ESPIPE
io_error,                           // EIO
is_a_directory,                     // EISDIR
message_size,                       // EMSGSIZE
network_down,                       // ENETDOWN
network_reset,                      // ENETRESET
network_unreachable,                // ENETUNREACH
no_buffer_space,                    // ENOBUFS
no_child_process,                   // ECHILD
no_link,                            // ENOLINK
no_lock_available,                  // ENOLCK
no_message_available,               // ENODATA
no_message,                         // ENOMSG
no_protocol_option,                 // ENOPROTOOPT
no_space_on_device,                 // ENOSPC
no_stream_resources,                // ENOSR
no_such_device_or_address,          // ENXIO
no_such_device,                     // ENODEV
no_such_file_or_directory,          // ENOENT
no_such_process,                    // ESRCH
not_a_directory,                    // ENOTDIR
not_a_socket,                       // ENOTSOCK
not_a_stream,                       // ENOSTR
not_connected,                      // ENOTCONN
not_enough_memory,                  // ENOMEM
not_supported,                      // ENOTSUP
operation_canceled,                 // ECANCELED
operation_in_progress,              // EINPROGRESS
operation_not_permitted,            // EPERM
operation_not_supported,            // EOPNOTSUPP
operation_would_block,              // EWOULDBLOCK
owner_dead,                         // EOWNERDEAD
permission_denied,                  // EACCES
protocol_error,                     // EPROTO
protocol_not_supported,             // EPROTONOSUPPORT
read_only_file_system,              // EROFS
resource_deadlock_would_occur,      // EDEADLK
resource_unavailable_try_again,     // EAGAIN
result_out_of_range,                // ERANGE
state_not_recoverable,              // ENOTRECOVERABLE
stream_timeout,                     // ETIME
```

```
    text_file_busy,                    // ETXTBSY
    timed_out,                         // ETIMEDOUT
    too_many_files_open_in_system,     // ENFILE
    too_many_files_open,               // EMFILE
    too_many_links,                    // EMLINK
    too_many_symbolic_link_levels,     // ELOOP
    value_too_large,                   // EOVERFLOW
    wrong_protocol_type,               // EPROTOTYPE
  };

  template <> struct is_error_condition_enum<errc> : true_type { }

  error_code make_error_code(errc e) noexcept;
  error_condition make_error_condition(errc e) noexcept;

  // 19.24.2 Comparison operators:
  bool operator==(const error_code& lhs, const error_code& rhs) noexcept;
  bool operator==(const error_code& lhs, const error_condition& rhs) noexcept;
  bool operator==(const error_condition& lhs, const error_code& rhs) noexcept;
  bool operator==(const error_condition& lhs, const error_condition& rhs) noexcept;
  bool operator!=(const error_code& lhs, const error_code& rhs) noexcept;
  bool operator!=(const error_code& lhs, const error_condition& rhs) noexcept;
  bool operator!=(const error_condition& lhs, const error_condition& rhs) noexcept;
  bool operator!=(const error_condition& lhs, const error_condition& rhs) noexcept;

  // 19.24.3 Hash support
  template <class T> struct hash;
  template <> struct hash<error_code>;
}  // namespace std
```

2 The value of each `enum errc` constant shall be the same as the value of the `<cerrno>` macro shown in the above synopsis. Whether or not the `<system_error>` implementation exposes the `<cerrno>` macros is unspecified.

1 The `is_error_code_enum` and `is_error_condition_enum` may be specialized for user-defined types to indicate that such types are eligible for `class error_code` and `class error_condition` automatic conversions, respectively.

19.16.1 Class `error_category` [syserr.errcat]

19.16.1.1 Class `error_category` overview [syserr.errcat.overview]

1 The class `error_category` serves as a base class for types used to identify the source and encoding of a particular category of error code. Classes may be derived from `error_category` to support categories of errors in addition to those defined in this International Standard. Such classes shall behave as specified in this subclause. [*Note:* `error_category` objects are passed by reference, and two such objects are equal if they have the same address. This means that applications using custom `error_category` types should create a single object of each such type. — *end note*]

```
namespace std {
  class error_category {
  public:
    constexpr error_category() noexcept;
    virtual ~error_category();
    error_category(const error_category&) = delete;
    error_category& operator=(const error_category&) = delete;
    virtual const char* name() const noexcept = 0;
```

```
    virtual error_condition default_error_condition(int ev) const noexcept;
    virtual bool equivalent(int code, const error_condition& condition) const noexcept;
    virtual bool equivalent(const error_code& code, int condition) const noexcept;
    virtual string message(int ev) const = 0;

    bool operator==(const error_category& rhs) const noexcept;
    bool operator!=(const error_category& rhs) const noexcept;
    bool operator<(const error_category& rhs) const noexcept;
  };

  const error_category& generic_category() noexcept;
  const error_category& system_category() noexcept;

}   // namespace std
```

19.16.1.2 Class error_category virtual members [syserr.errcat.virtuals]

`virtual ~error_category();`

1 *Effects:* Destroys an object of class `error_category`.

`virtual const char* name() const noexcept = 0;`

1 *Returns:* A string naming the error category.

`virtual error_condition default_error_condition(int ev) const noexcept;`

2 *Returns:* `error_condition(ev, *this)`.

`virtual bool equivalent(int code, const error_condition& condition) const noexcept;`

3 *Returns:* `default_error_condition(code) == condition`.

`virtual bool equivalent(const error_code& code, int condition) const noexcept;`

4 *Returns:* `*this == code.category() && code.value() == condition`.

`virtual string message(int ev) const = 0;`

5 *Returns:* A string that describes the error condition denoted by `ev`.

19.19.0.1 Class error_category non-virtual members [syserr.errcat.nonvirtuals]

`constexpr error_category() noexcept;`

1 *Effects:* Constructs an object of class `error_category`.

`bool operator==(const error_category& rhs) const noexcept;`

2 *Returns:* `this == &rhs`.

`bool operator!=(const error_category& rhs) const noexcept;`

3 *Returns:* `!(*this == rhs)`.

`bool operator<(const error_category& rhs) const noexcept;`

4 *Returns:* `less<const error_category*>()(this, &rhs)`.

[*Note:* `less` (20.339.3) provides a total ordering for pointers. — *end note*]

19.19.0.2 Program defined classes derived from `error_category` [syserr.errcat.derived]

`virtual const char* name() const noexcept = 0;`

1 *Returns:* A string naming the error category.

`virtual error_condition default_error_condition(int ev) const noexcept;`

2 *Returns:* An object of type `error_condition` that corresponds to `ev`.

`virtual bool equivalent(int code, const error_condition& condition) const noexcept;`

3 *Returns:* `true` if, for the category of error represented by `*this`, `code` is considered equivalent to `condition`; otherwise, `false`.

`virtual bool equivalent(const error_code& code, int condition) const noexcept;`

4 *Returns:* `true` if, for the category of error represented by `*this`, `code` is considered equivalent to `condition`; otherwise, `false`.

19.19.0.3 Error category objects [syserr.errcat.objects]

`const error_category& generic_category() noexcept;`

1 *Returns:* A reference to an object of a type derived from class `error_category`. All calls to this function shall return references to the same object.

2 *Remarks:* The object's `default_error_condition` and `equivalent` virtual functions shall behave as specified for the class `error_category`. The object's `name` virtual function shall return a pointer to the string `"generic"`.

`const error_category& system_category() noexcept;`

3 *Returns:* A reference to an object of a type derived from class `error_category`. All calls to this function shall return references to the same object.

4 *Remarks:* The object's `equivalent` virtual functions shall behave as specified for class `error_category`. The object's `name` virtual function shall return a pointer to the string `"system"`. The object's `default_error_condition` virtual function shall behave as follows:

If the argument `ev` corresponds to a POSIX `errno` value `posv`, the function shall return `error_condition(posv, generic_category())`. Otherwise, the function shall return `error_condition(ev, system_category())`. What constitutes correspondence for any given operating system is unspecified. [*Note:* The number of potential system error codes is large and unbounded, and some may not correspond to any POSIX `errno` value. Thus implementations are given latitude in determining correspondence. — *end note*]

19.22.1 Class `error_code` [syserr.errcode]

19.22.1.1 Class `error_code` overview [syserr.errcode.overview]

1 The class `error_code` describes an object used to hold error code values, such as those originating from the operating system or other low-level application program interfaces. [*Note:* Class `error_code` is an adjunct to error reporting by exception. — *end note*]

```
namespace std {
  class error_code {
  public:
    // 19.22.1.2 constructors:
```

```
    error_code() noexcept;
    error_code(int val, const error_category& cat) noexcept;
    template <class ErrorCodeEnum>
      error_code(ErrorCodeEnum e) noexcept;

    // 19.23.0.1 modifiers:
    void assign(int val, const error_category& cat) noexcept;
    template <class ErrorCodeEnum>
        error_code& operator=(ErrorCodeEnum e) noexcept;
    void clear() noexcept;

    // 19.23.0.2 observers:
    int value() const noexcept;
    const error_category& category() const noexcept;
    error_condition default_error_condition() const noexcept;
    string message() const;
    explicit operator bool() const noexcept;

  private:
    int val_;                    // exposition only
    const error_category* cat_; // exposition only
  };

  // 19.24.0.1 non-member functions:
  error_code make_error_code(errc e) noexcept;
  bool operator<(const error_code& lhs, const error_code& rhs) noexcept;

  template <class charT, class traits>
    basic_ostream<charT,traits>&
      operator<<(basic_ostream<charT,traits>& os, const error_code& ec);
}   // namespace std
```

19.22.1.2 Class `error_code` constructors [syserr.errcode.constructors]

`error_code() noexcept;`

1 *Effects:* Constructs an object of type `error_code`.

2 *Postconditions:* `val_ == 0` and `cat_ == &system_category()`.

`error_code(int val, const error_category& cat) noexcept;`

3 *Effects:* Constructs an object of type `error_code`.

4 *Postconditions:* `val_ == val` and `cat_ == &cat`.

```
template <class ErrorCodeEnum>
  error_code(ErrorCodeEnum e) noexcept;
```

5 *Effects:* Constructs an object of type `error_code`.

6 *Postconditions:* `*this == make_error_code(e)`.

7 *Remarks:* This constructor shall not participate in overload resolution unless `is_error_code_enum<ErrorCodeEnum>::value` is `true`.

19.23.0.1 Class error_code modifiers [syserr.errcode.modifiers]

```
void assign(int val, const error_category& cat) noexcept;
```

1 *Postconditions:* `val_ == val` and `cat_ == &cat`.

```
template <class ErrorCodeEnum>
    error_code& operator=(ErrorCodeEnum e) noexcept;
```

2 *Postconditions:* `*this == make_error_code(e)`.

3 *Returns:* `*this`.

4 *Remarks:* This operator shall not participate in overload resolution unless `is_error_code_enum<ErrorCodeEnum>::value` is `true`.

```
void clear() noexcept;
```

5 *Postconditions:* `value() == 0` and `category() == system_category()`.

19.23.0.2 Class error_code observers [syserr.errcode.observers]

```
int value() const noexcept;
```

1 *Returns:* `val_`.

```
const error_category& category() const noexcept;
```

2 *Returns:* `*cat_`.

```
error_condition default_error_condition() const noexcept;
```

3 *Returns:* `category().default_error_condition(value())`.

```
string message() const;
```

4

Returns: `category().message(value())`.

```
explicit operator bool() const noexcept;
```

1 *Returns:* `value() != 0`.

19.24.0.1 Class error_code non-member functions [syserr.errcode.nonmembers]

```
error_code make_error_code(errc e) noexcept;
```

1 *Returns:* `error_code(static_cast<int>(e), generic_category())`.

```
bool operator<(const error_code& lhs, const error_code& rhs) noexcept;
```

2 *Returns:* `lhs.category() < rhs.category() || lhs.category() == rhs.category() && lhs.value() < rhs.value()`.

```
template <class charT, class traits>
  basic_ostream<charT,traits>&
    operator<<(basic_ostream<charT,traits>& os, const error_code& ec);
```

3 *Effects:* `os << ec.category().name() << ':'  << ec.value()`.

19.24.1 Class error_condition [syserr.errcondition]

19.24.1.1 Class error_condition overview [syserr.errcondition.overview]

1 The class error_condition describes an object used to hold values identifying error conditions. [*Note:* error_condition values are portable abstractions, while error_code values (19.22.1) are implementation specific. — *end note*]

```
namespace std {
  class error_condition {
  public:
    // 19.24.1.2 constructors:
    error_condition() noexcept;
    error_condition(int val, const error_category& cat) noexcept;
    template <class ErrorConditionEnum>
      error_condition(ErrorConditionEnum e) noexcept;

    // 19.24.1.3 modifiers:
    void assign(int val, const error_category& cat) noexcept;
    template<class ErrorConditionEnum>
        error_condition& operator=(ErrorConditionEnum e) noexcept;
    void clear() noexcept;

    // 19.24.1.4 observers:
    int value() const noexcept;
    const error_category& category() const noexcept;
    string message() const;
    explicit operator bool() const noexcept;

  private:
    int val_;                    // exposition only
    const error_category* cat_;  // exposition only
  };

  // 19.24.1.5 non-member functions:
  bool operator<(const error_condition& lhs, const error_condition& rhs) noexcept;
} // namespace std
```

19.24.1.2 Class error_condition constructors [syserr.errcondition.constructors]

`error_condition() noexcept;`

1 *Effects:* Constructs an object of type error_condition.

2 *Postconditions:* val_ == 0 and cat_ == &generic_category().

`error_condition(int val, const error_category& cat) noexcept;`

3 *Effects:* Constructs an object of type error_condition.

4 *Postconditions:* val_ == val and cat_ == &cat.

```
template <class ErrorConditionEnum>
  error_condition(ErrorConditionEnum e) noexcept;
```

5 *Effects:* Constructs an object of type error_condition.

6 *Postcondition:* *this == make_error_condition(e).

7 *Remarks:* This constructor shall not participate in overload resolution unless is_error_condition_enum<ErrorConditionEnum>::value is true.

19.24.1.3 Class error_condition modifiers [syserr.errcondition.modifiers]

```
void assign(int val, const error_category& cat) noexcept;
```

1 *Postconditions:* `val_ == val` and `cat_ == &cat`.

```
template <class ErrorConditionEnum>
    error_condition& operator=(ErrorConditionEnum e) noexcept;
```

2 *Postcondition:* `*this == make_error_condition(e)`.

3 *Returns:* `*this`.

4 *Remarks:* This operator shall not participate in overload resolution unless `is_error_condition_enum<ErrorConditionEnum>::value` is `true`.

```
void clear() noexcept;
```

Postconditions: `value() == 0` and `category() == generic_category()`.

19.24.1.4 Class error_condition observers [syserr.errcondition.observers]

```
int value() const noexcept;
```

1 *Returns:* `val_`.

```
const error_category& category() const noexcept;
```

2 *Returns:* `*cat_`.

```
string message() const;
```

3 *Returns:* `category().message(value())`.

```
explicit operator bool() const noexcept;
```

4 *Returns:* `value() != 0`.

19.24.1.5 Class error_condition non-member functions [syserr.errcondition.nonmembers]

```
error_condition make_error_condition(errc e) noexcept;
```

Returns: `error_condition(static_cast<int>(e), generic_category())`.

```
bool operator<(const error_condition& lhs, const error_condition& rhs) noexcept;
```

1 *Returns:* `lhs.category() < rhs.category() || lhs.category() == rhs.category() && lhs.value() < rhs.value()`.

19.24.2 Comparison operators [syserr.compare]

```
bool operator==(const error_code& lhs, const error_code& rhs) noexcept;
```

1 *Returns:* `lhs.category() == rhs.category() && lhs.value() == rhs.value()`.

```
bool operator==(const error_code& lhs, const error_condition& rhs) noexcept;
```

2 *Returns:* `lhs.category().equivalent(lhs.value(), rhs) || rhs.category().equivalent(lhs, rhs.value())`.

```
bool operator==(const error_condition& lhs, const error_code& rhs) noexcept;
```

3 *Returns:* `rhs.category().equivalent(rhs.value(), lhs) || lhs.category().equivalent(rhs, lhs.value())`.

```
bool operator==(const error_condition& lhs, const error_condition& rhs) noexcept;
```

4 *Returns:* `lhs.category() == rhs.category() && lhs.value() == rhs.value()`.

```
bool operator!=(const error_code& lhs, const error_code& rhs) noexcept;
bool operator!=(const error_code& lhs, const error_condition& rhs) noexcept;
bool operator!=(const error_condition& lhs, const error_code& rhs) noexcept;
bool operator!=(const error_condition& lhs, const error_condition& rhs) noexcept;
```

5 *Returns:* `!(lhs == rhs)`.

19.24.3 System error hash support [syserr.hash]

```
template <> struct hash<error_code>;
```

1 The template specialization shall meet the requirements of class template `hash` (20.443.1).

19.24.4 Class `system_error` [syserr.syserr]

19.24.4.1 Class `system_error` overview [syserr.syserr.overview]

1

The class `system_error` describes an exception object used to report error conditions that have an associated error code. Such error conditions typically originate from the operating system or other low-level application program interfaces.

1 [*Note:* If an error represents an out-of-memory condition, implementations are encouraged to throw an exception object of type `bad_alloc` 18.223.1.1 rather than `system_error`. — *end note*]

```
namespace std {
  class system_error : public runtime_error {
  public:
    system_error(error_code ec, const string& what_arg);
    system_error(error_code ec, const char* what_arg);
    system_error(error_code ec);
    system_error(int ev, const error_category& ecat,
        const string& what_arg);
    system_error(int ev, const error_category& ecat,
        const char* what_arg);
    system_error(int ev, const error_category& ecat);
    const error_code& code() const noexcept;
    const char* what() const noexcept;
  };
}   // namespace std
```

19.25.0.1 Class `system_error` members [syserr.syserr.members]

```
system_error(error_code ec, const string& what_arg);
```

1 *Effects:* Constructs an object of class `system_error`.

2 *Postconditions:* `code() == ec`.

`string(what()).find(what_arg) != string::npos`.

```
system_error(error_code ec, const char* what_arg);
```

3 *Effects:* Constructs an object of class `system_error`.

4 *Postconditions:* `code() == ec`.

`string(what()).find(what_arg) != string::npos`.

```
system_error(error_code ec);
```

5 *Effects:* Constructs an object of class `system_error`.

6 *Postconditions:* `code() == ec`.

```
system_error(int ev, const error_category& ecat,
  const string& what_arg);
```

7 *Effects:* Constructs an object of class `system_error`.

8 *Postconditions:* `code() == error_code(ev, ecat)`.

`string(what()).find(what_arg) != string::npos`.

```
system_error(int ev, const error_category& ecat,
  const char* what_arg);
```

9 *Effects:* Constructs an object of class `system_error`.

10 *Postconditions:* `code() == error_code(ev, ecat)`.

`string(what()).find(what_arg) != string::npos`.

```
system_error(int ev, const error_category& ecat);
```

11 *Effects:* Constructs an object of class `system_error`.

12 *Postconditions:* `code() == error_code(ev, ecat)`.

```
const error_code& code() const noexcept;
```

13 *Returns:* `ec` or `error_code(ev, ecat)`, from the constructor, as appropriate.

```
const char* what() const noexcept;
```

14

Returns: An NTBS incorporating the arguments supplied in the constructor.

[*Note:* The returned NTBS might be the contents of `what_arg + ":  " + code.message()`. — *end note*]

20 General utilities library [utilities]

20.1 General [utilities.general]

1 This Clause describes utilities that are generally useful in C++ programs; some of these utilities are used by other elements of the C++ standard library. These utilities are summarized in Table 44.

Table 44 — General utilities library summary

Subclause		Header(s)
20.2	Utility components	`<utility>`
20.15	Pairs	`<utility>`
20.34	Tuples	`<tuple>`
20.66	Compile-time integer sequences	`<utility>`
20.68	Fixed-size sequences of bits	`<bitset>`

20.82	Memory	`<memory>`
		`<cstdlib>`
		`<cstring>`

20.127	Smart pointers	`<memory>`
20.283	Function objects	`<functional>`
20.449	Type traits	`<type_traits>`
20.575	Compile-time rational arithmetic	`<ratio>`
20.578	Time utilities	`<chrono>` `<ctime>`
20.606	Scoped allocators	`<scoped_allocator>`
20.610	Type indexes	`<typeindex>`

20.2 Utility components [utility]

1

This subclause contains some basic function and class templates that are used throughout the rest of the library.

Header `<utility>` synopsis

1 The header `<utility>` defines several types and function templates that are described in this Clause. It also defines the template `pair` and various function templates that operate on `pair` objects.

```
#include <initializer_list>

namespace std {
  // 20.3.1, operators:
  namespace rel_ops {
    template<class T> bool operator!=(const T&, const T&);
    template<class T> bool operator> (const T&, const T&);
    template<class T> bool operator<=(const T&, const T&);
    template<class T> bool operator>=(const T&, const T&);
  }

  // 20.3.2, swap:
```

```
template<class T> void swap(T& a, T& b) noexcept(see below);
template <class T, size_t N> void swap(T (&a)[N], T (&b)[N]) noexcept(noexcept(swap(*a, *b)));

// 20.4.1, exchange:
template <class T, class U=T> T exchange(T& obj, U&& new_val);

// 20.4.2, forward/move:
template <class T>
  constexpr T&& forward(remove_reference_t<T>& t) noexcept;
template <class T>
  constexpr T&& forward(remove_reference_t<T>&& t) noexcept;
template <class T>
  constexpr remove_reference_t<T>&& move(T&&) noexcept;
template <class T>
  constexpr conditional_t<
  !is_nothrow_move_constructible<T>::value && is_copy_constructible<T>::value,
  const T&, T&&> move_if_noexcept(T& x) noexcept;

// 20.11.1, declval:
template <class T>
  add_rvalue_reference_t<T> declval() noexcept;  // as unevaluated operand

// 20.15, pairs:
template <class T1, class T2> struct pair;

// 20.28.1, pair specialized algorithms:
template <class T1, class T2>
  constexpr bool operator==(const pair<T1,T2>&, const pair<T1,T2>&);
template <class T1, class T2>
  constexpr bool operator< (const pair<T1,T2>&, const pair<T1,T2>&);
template <class T1, class T2>
  constexpr bool operator!=(const pair<T1,T2>&, const pair<T1,T2>&);
template <class T1, class T2>
  constexpr bool operator> (const pair<T1,T2>&, const pair<T1,T2>&);
template <class T1, class T2>
  constexpr bool operator>=(const pair<T1,T2>&, const pair<T1,T2>&);
template <class T1, class T2>
  constexpr bool operator<=(const pair<T1,T2>&, const pair<T1,T2>&);
template <class T1, class T2>
  void swap(pair<T1,T2>& x, pair<T1,T2>& y) noexcept(noexcept(x.swap(y)));
template <class T1, class T2>
  constexpr see below make_pair(T1&&, T2&&);

// 20.33.1, tuple-like access to pair:
template <class T> class tuple_size;
template <size_t I, class T> class tuple_element;

template <class T1, class T2> struct tuple_size<pair<T1, T2> >;
template <class T1, class T2> struct tuple_element<0, pair<T1, T2> >;
template <class T1, class T2> struct tuple_element<1, pair<T1, T2> >;

template<size_t I, class T1, class T2>
  constexpr tuple_element_t<I, pair<T1, T2>>&
    get(pair<T1, T2>&) noexcept;
template<size_t I, class T1, class T2>
```

```
    constexpr tuple_element_t<I, pair<T1, T2>>&&
      get(pair<T1, T2>&&) noexcept;
  template<size_t I, class T1, class T2>
    constexpr const tuple_element_t<I, pair<T1, T2>>&
      get(const pair<T1, T2>&) noexcept;
  template <class T, class U>
    constexpr T& get(pair<T, U>& p) noexcept;
  template <class T, class U>
    constexpr const T& get(const pair<T, U>& p) noexcept;
  template <class T, class U>
    constexpr T&& get(pair<T, U>&& p) noexcept;
  template <class T, class U>
    constexpr T& get(pair<U, T>& p) noexcept;
  template <class T, class U>
    constexpr const T& get(const pair<U, T>& p) noexcept;
  template <class T, class U>
    constexpr T&& get(pair<U, T>&& p) noexcept;

  // 20.33.2, pair piecewise construction
  struct piecewise_construct_t { };
  constexpr piecewise_construct_t piecewise_construct{};
  template <class... Types> class tuple;  // defined in <tuple>

  // 20.66, Compile-time integer sequences
  template<class T, T...> struct integer_sequence;
  template<size_t... I>
    using index_sequence = integer_sequence<size_t, I...>;

  template<class T, T N>
    using make_integer_sequence = integer_sequence<T, see below>;
  template<size_t N>
    using make_index_sequence = make_integer_sequence<size_t, N>;

  template<class... T>
    using index_sequence_for = make_index_sequence<sizeof...(T)>;
}
```

20.3.1 Operators [operators]

1 To avoid redundant definitions of `operator!=` out of `operator==` and operators `>`, `<=`, and `>=` out of `operator<`, the library provides the following:

```
template <class T> bool operator!=(const T& x, const T& y);
```

2 *Requires:* Type `T` is `EqualityComparable` (Table 17).

3 *Returns:* `!(x == y)`.

```
template <class T> bool operator>(const T& x, const T& y);
```

4 *Requires:* Type `T` is `LessThanComparable` (Table 18).

5 *Returns:* `y < x`.

```
template <class T> bool operator<=(const T& x, const T& y);
```

6 *Requires:* Type `T` is `LessThanComparable` (Table 18).

7 *Returns:* `!(y < x)`.

```
template <class T> bool operator>=(const T& x, const T& y);
```

8 *Requires:* Type `T` is `LessThanComparable` (Table 18).

9 *Returns:* `!(x < y)`.

10 In this library, whenever a declaration is provided for an `operator!=`, `operator>`, `operator>=`, or `operator<=`, and requirements and semantics are not explicitly provided, the requirements and semantics are as specified in this Clause.

20.3.2 swap [utility.swap]

```
template<class T> void swap(T& a, T& b) noexcept(see below);
```

1

Remark: The expression inside `noexcept` is equivalent to:

```
is_nothrow_move_constructible<T>::value &&
is_nothrow_move_assignable<T>::value
```

1 *Requires:* Type `T` shall be `MoveConstructible` (Table 20) and `MoveAssignable` (Table 22).

2 *Effects:* Exchanges values stored in two locations.

```
template<class T, size_t N>
  void swap(T (&a)[N], T (&b)[N]) noexcept(noexcept(swap(*a, *b)));
```

3 *Requires:* `a[i]` shall be swappable with (17.143.0.1) `b[i]` for all `i` in the range `[0,N)`.

4 *Effects:* `swap_ranges(a, a + N, b)`

20.4.1 exchange [utility.exchange]

```
template <class T, class U=T> T exchange(T& obj, U&& new_val);
```

1 *Effects:* Equivalent to:

```
T old_val = std::move(obj);
obj = std::forward<U>(new_val);
return old_val;
```

20.4.2 forward/move helpers [forward]

1

The library provides templated helper functions to simplify applying move semantics to an lvalue and to simplify the implementation of forwarding functions.

```
template <class T> constexpr T&& forward(remove_reference_t<T>& t) noexcept;
template <class T> constexpr T&& forward(remove_reference_t<T>&& t) noexcept;
```

1 *Returns:* `static_cast<T&&>(t)`.

2

Remark: If the second form is instantiated with an lvalue reference type, the program is ill-formed.

1 [*Example:*

```
template <class T, class A1, class A2>
shared_ptr<T> factory(A1&& a1, A2&& a2) {
  return shared_ptr<T>(new T(std::forward<A1>(a1), std::forward<A2>(a2)));
}
```

```
struct A {
  A(int&, const double&);
};

void g() {
  shared_ptr<A> sp1 = factory<A>(2, 1.414); // error: 2 will not bind to int&
  int i = 2;
  shared_ptr<A> sp2 = factory<A>(i, 1.414); // OK
}
```

2

In the first call to `factory`, `A1` is deduced as `int`, so 2 is forwarded to `A`'s constructor as an rvalue. In the second call to `factory`, `A1` is deduced as `int&`, so `i` is forwarded to `A`'s constructor as an lvalue. In both cases, `A2` is deduced as `double`, so 1.414 is forwarded to `A`'s constructor as an rvalue.

— *end example*]

```
template <class T> constexpr remove_reference_t<T>&& move(T&& t) noexcept;
```

1 *Returns:* `static_cast<remove_reference_t<T>&&>(t)`.

2 [*Example:*

```
template <class T, class A1>
shared_ptr<T> factory(A1&& a1) {
  return shared_ptr<T>(new T(std::forward<A1>(a1)));
}

struct A {
  A();
  A(const A&);   // copies from lvalues
  A(A&&);        // moves from rvalues
};

void g() {
  A a;
  shared_ptr<A> sp1 = factory<A>(a);              // "a" binds to A(const A&)
  shared_ptr<A> sp1 = factory<A>(std::move(a));   // "a" binds to A(A&&)
}
```

3 In the first call to `factory`, `A1` is deduced as `A&`, so `a` is forwarded as a non-const lvalue. This binds to the constructor `A(const A&)`, which copies the value from `a`. In the second call to `factory`, because of the call `std::move(a)`, `A1` is deduced as `A`, so `a` is forwarded as an rvalue. This binds to the constructor `A(A&&)`, which moves the value from `a`.

— *end example*]

```
template <class T> constexpr conditional_t<
  !is_nothrow_move_constructible<T>::value && is_copy_constructible<T>::value,
  const T&, T&&> move_if_noexcept(T& x) noexcept;
```

4 *Returns:* `std::move(x)`

20.11.1 Function template `declval` [declval]

1

The library provides the function template `declval` to simplify the definition of expressions which occur as unevaluated operands (Clause 5).

```
template <class T>
  add_rvalue_reference_t<T> declval() noexcept;  // as unevaluated operand
```

1 *Remarks:* If this function is odr-used (3.91), the program is ill-formed.

2 *Remarks:* The template parameter `T` of `declval` may be an incomplete type.

[*Example:*

```
template <class To, class From>
  decltype(static_cast<To>(declval<From>())) convert(From&&);
```

declares a function template `convert` which only participates in overloading if the type `From` can be explicitly converted to type `To`. For another example see class template `common_type` (20.570.1.6). — *end example*]

20.15 Pairs [pairs]

20.15.1 In general [pairs.general]

1 The library provides a template for heterogeneous pairs of values. The library also provides a matching function template to simplify their construction and several templates that provide access to `pair` objects as if they were `tuple` objects (see 20.56.0.1 and 20.57.0.1).

20.15.2 Class template `pair` [pairs.pair]

```
// defined in header <utility>

namespace std {
  template <class T1, class T2>
  struct pair {
    typedef T1 first_type;
    typedef T2 second_type;

    T1 first;
    T2 second;
    pair(const pair&) = default;
    pair(pair&&) = default;
    constexpr pair();
    constexpr pair(const T1& x, const T2& y);
    template<class U, class V> constexpr pair(U&& x, V&& y);
    template<class U, class V> constexpr pair(const pair<U, V>& p);
    template<class U, class V> constexpr pair(pair<U, V>&& p);
    template <class... Args1, class... Args2>
      pair(piecewise_construct_t,
           tuple<Args1...> first_args, tuple<Args2...> second_args);

    pair& operator=(const pair& p);
    template<class U, class V> pair& operator=(const pair<U, V>& p);
    pair& operator=(pair&& p) noexcept(see below);
    template<class U, class V> pair& operator=(pair<U, V>&& p);

    void swap(pair& p) noexcept(see below);
  };
}
```

1 Constructors and member functions of `pair` shall not throw exceptions unless one of the element-wise operations specified to be called for that operation throws an exception.

2

The defaulted move and copy constructor, respectively, of pair shall be a `constexpr` function if and only if all required element-wise initializations for copy and move, respectively, would satisfy the requirements for a `constexpr` function.

```
constexpr pair();
```

1 *Requires:* `is_default_constructible<first_type>::value` is `true` and `is_default_constructible<second_type>::value` is `true`.

2 *Effects:* Value-initializes `first` and `second`.

```
constexpr pair(const T1& x, const T2& y);
```

3 *Requires:* `is_copy_constructible<first_type>::value` is `true` and `is_copy_constructible<second_type>::value` is `true`.

4

Effects: The constructor initializes `first` with `x` and `second` with `y`.

```
template<class U, class V> constexpr pair(U&& x, V&& y);
```

1

Requires: `is_constructible<first_type, U&&>::value` is `true` and `is_constructible<second_type, V&&>::value` is `true`.

1

Effects: The constructor initializes `first` with `std::forward<U>(x)` and `second` with `std::forward<V>(y)`.

1

Remarks: If `U` is not implicitly convertible to `first_type` or `V` is not implicitly convertible to `second_type` this constructor shall not participate in overload resolution.

```
template<class U, class V> constexpr pair(const pair<U, V>& p);
```

1 *Requires:* `is_constructible<first_type, const U&>::value` is `true` and `is_constructible<second_type, const V&>::value` is `true`.

2 *Effects:* Initializes members from the corresponding members of the argument.

3

Remark: This constructor shall not participate in overload resolution unless `const U&` is implicitly convertible to `first_type` and `const V&` is implicitly convertible to `second_type`.

```
template<class U, class V> constexpr pair(pair<U, V>&& p);
```

1 *Requires:* `is_constructible<first_type, U&&>::value` is `true` and `is_constructible<second_type, V&&>::value` is `true`.

2 *Effects:* The constructor initializes `first` with `std::forward<U>(p.first)` and `second` with `std::forward<V>(p.second)`.

3 *Remark:* This constructor shall not participate in overload resolution unless `U` is implicitly convertible to `first_type` and `V` is implicitly convertible to `second_type`.

```
template<class... Args1, class... Args2>
  pair(piecewise_construct_t,
       tuple<Args1...> first_args, tuple<Args2...> second_args);
```

4 *Requires:* `is_constructible<first_type, Args1&&...>::value` is `true` and `is_constructible<second_type, Args2&&...>::value` is `true`.

5 *Effects:* The constructor initializes `first` with arguments of types `Args1...` obtained by forwarding the elements of `first_args` and initializes `second` with arguments of types `Args2...` obtained by forwarding the elements of `second_args`. (Here, forwarding an element `x` of type `U` within a `tuple` object means calling `std::forward<U>(x)`.) This form of construction, whereby constructor arguments for `first` and `second` are each provided in a separate `tuple` object, is called *piecewise construction*.

```
pair& operator=(const pair& p);
```

6 *Requires:* `is_copy_assignable<first_type>::value` is `true` and `is_copy_assignable<second_type>::value` is `true`.

7 *Effects:* Assigns `p.first` to `first` and `p.second` to `second`.

8 *Returns:* `*this`.

```
template<class U, class V> pair& operator=(const pair<U, V>& p);
```

9

Requires: `is_assignable<first_type&, const U&>::value` is `true` and `is_assignable<second_type&, const V&>::value` is `true`.

1 *Effects:* Assigns `p.first` to `first` and `p.second` to `second`.

2 *Returns:* `*this`.

```
pair& operator=(pair&& p) noexcept(see below);
```

3 *Remarks:* The expression inside `noexcept` is equivalent to:

```
is_nothrow_move_assignable<T1>::value &&
is_nothrow_move_assignable<T2>::value
```

4 *Requires:* `is_move_assignable<first_type>::value` is `true` and `is_move_assignable<second_type>::value` is `true`.

5 *Effects:* Assigns to `first` with `std::forward<first_type>(p.first)` and to `second` with `std::forward<second_type>(p.second)`.

6 *Returns:* `*this`.

```
template<class U, class V> pair& operator=(pair<U, V>&& p);
```

7 *Requires:* `is_assignable<first_type&, U&&>::value` is `true` and `is_assignable<second_type&, V&&>::value` is `true`.

8 *Effects:* Assigns to `first` with `std::forward<U>(p.first)` and to `second` with `std::forward<V>(p.second)`.

9 *Returns:* `*this`.

```
void swap(pair& p) noexcept(see below);
```

10 *Remarks:* The expression inside `noexcept` is equivalent to:

```
noexcept(swap(first, p.first)) &&
noexcept(swap(second, p.second))
```

11 *Requires:* `first` shall be swappable with (17.143.0.1) `p.first` and `second` shall be swappable with `p.second`.

12 *Effects:* Swaps `first` with `p.first` and `second` with `p.second`.

20.28.1 Specialized algorithms [pairs.spec]

```
template <class T1, class T2>
  constexpr bool operator==(const pair<T1, T2>& x, const pair<T1, T2>& y);
```

1 *Returns:* `x.first == y.first && x.second == y.second`.

```
template <class T1, class T2>
  constexpr bool operator<(const pair<T1, T2>& x, const pair<T1, T2>& y);
```

2 *Returns:* `x.first < y.first || (!(y.first < x.first) && x.second < y.second)`.

```
template <class T1, class T2>
  constexpr bool operator!=(const pair<T1, T2>& x, const pair<T1, T2>& y);
```

1 *Returns:* `!(x == y)`

```
template <class T1, class T2>
  constexpr bool operator>(const pair<T1, T2>& x, const pair<T1, T2>& y);
```

2 *Returns:* `y < x`

```
template <class T1, class T2>
  constexpr bool operator>=(const pair<T1, T2>& x, const pair<T1, T2>& y);
```

3 *Returns:* `!(x < y)`

```
template <class T1, class T2>
  constexpr bool operator<=(const pair<T1, T2>& x, const pair<T1, T2>& y);
```

4 *Returns:* `!(y < x)`

```
template<class T1, class T2> void swap(pair<T1, T2>& x, pair<T1, T2>& y)
  noexcept(noexcept(x.swap(y)));
```

5 *Effects:* `x.swap(y)`

```
template <class T1, class T2>
  constexpr pair<V1, V2> make_pair(T1&& x, T2&& y);
```

1 *Returns:* `pair<V1, V2>(std::forward<T1>(x), std::forward<T2>(y))`;

where `V1` and `V2` are determined as follows: Let `Ui` be `decay_t<Ti>` for each `Ti`. Then each `Vi` is `X&` if `Ui` equals `reference_wrapper<X>`, otherwise `Vi` is `Ui`.

1 [*Example:* In place of:

```
return pair<int, double>(5, 3.1415926);    // explicit types
```

a C++ program may contain:

```
return make_pair(5, 3.1415926);            // types are deduced
```

— *end example*]

20.33.1 Tuple-like access to pair [pair.astuple]

```
template <class T1, class T2>
struct tuple_size<pair<T1, T2>>
  : integral_constant<size_t, 2> { };
```

```
tuple_element<0, pair<T1, T2> >::type
```

1 *Value:* the type `T1`.

```
tuple_element<1, pair<T1, T2> >::type
```

2 *Value:* the type T2.

```
template<size_t I, class T1, class T2>
  constexpr tuple_element_t<I, pair<T1, T2>>&
    get(pair<T1, T2>& p) noexcept;
template<size_t I, class T1, class T2>
  constexpr const tuple_element_t<I, pair<T1, T2>>&
    get(const pair<T1, T2>& p) noexcept;
```

3 *Returns:* If `I == 0` returns `p.first`; if `I == 1` returns `p.second`; otherwise the program is ill-formed.

```
template<size_t I, class T1, class T2>
  constexpr tuple_element_t<I, pair<T1, T2>>&&
    get(pair<T1, T2>&& p) noexcept;
```

4 *Returns:* If `I == 0` returns `std::forward<T1&&>(p.first)`; if `I == 1` returns `std::forward<T2&&>(p.second)`; otherwise the program is ill-formed.

```
template <class T, class U>
  constexpr T& get(pair<T, U>& p) noexcept;
template <class T, class U>
  constexpr const T& get(const pair<T, U>& p) noexcept;
```

5 *Requires:* `T` and `U` are distinct types. Otherwise, the program is ill-formed.

6 *Returns:* `get<0>(p);`

```
template <class T, class U>
  constexpr T&& get(pair<T, U>&& p) noexcept;
```

7 *Requires:* `T` and `U` are distinct types. Otherwise, the program is ill-formed.

8 *Returns:* `get<0>(std::move(p));`

```
template <class T, class U>
  constexpr T& get(pair<U, T>& p) noexcept;
template <class T, class U>
  constexpr const T& get(const pair<U, T>& p) noexcept;
```

9 *Requires:* `T` and `U` are distinct types. Otherwise, the program is ill-formed.

10 *Returns:* `get<1>(p);`

```
template <class T, class U>
  constexpr T&& get(pair<U, T>&& p) noexcept;
```

11 *Requires:* `T` and `U` are distinct types. Otherwise, the program is ill-formed.

12 *Returns:* `get<1>(std::move(p));`

20.33.2 Piecewise construction [pair.piecewise]

```
struct piecewise_construct_t { };
constexpr piecewise_construct_t piecewise_construct{};
```

1 The `struct piecewise_construct_t` is an empty structure type used as a unique type to disambiguate constructor and function overloading. Specifically, `pair` has a constructor with `piecewise_construct_t` as the first argument, immediately followed by two `tuple` (20.34) arguments used for piecewise construction of the elements of the `pair` object.

20.34 Tuples [tuple]

20.34.1 In general [tuple.general]

1

This subclause describes the tuple library that provides a tuple type as the class template `tuple` that can be instantiated with any number of arguments. Each template argument specifies the type of an element in the `tuple`. Consequently, tuples are heterogeneous, fixed-size collections of values. An instantiation of `tuple` with two arguments is similar to an instantiation of `pair` with the same two arguments. See 20.15.

1 **Header `<tuple>` synopsis**

```
namespace std {
  // 20.37.1, class template tuple:
  template <class... Types> class tuple;

  // 20.53.0.2, tuple creation functions:
  const unspecified  ignore;

  template <class... Types>
    constexpr tuple<VTypes...> make_tuple(Types&&...);
  template <class... Types>
    constexpr tuple<Types&&...> forward_as_tuple(Types&&...) noexcept;

  template<class... Types>
    constexpr tuple<Types&...> tie(Types&...) noexcept;

  template <class... Tuples>
    constexpr tuple<Ctypes...> tuple_cat(Tuples&&...);

  // 20.56.0.1, tuple helper classes:
  template <class T> class tuple_size;  // undefined
  template <class T> class tuple_size<const T>;
  template <class T> class tuple_size<volatile T>;
  template <class T> class tuple_size<const volatile T>;

  template <class... Types> class tuple_size<tuple<Types...> >;

  template <size_t I, class T> class tuple_element;    // undefined
  template <size_t I, class T> class tuple_element<I, const T>;
  template <size_t I, class T> class tuple_element<I, volatile T>;
  template <size_t I, class T> class tuple_element<I, const volatile T>;

  template <size_t I, class... Types> class tuple_element<I, tuple<Types...> >;

  template <size_t I, class T>
    using tuple_element_t = typename tuple_element<I, T>::type;
```

```
  // 20.57.0.1, element access:
  template <size_t I, class... Types>
    constexpr tuple_element_t<I, tuple<Types...>>&
      get(tuple<Types...>&) noexcept;
  template <size_t I, class... Types>
    constexpr tuple_element_t<I, tuple<Types...>>&&
      get(tuple<Types...>&&) noexcept;
  template <size_t I, class... Types>
    constexpr const tuple_element_t<I, tuple<Types...>>&
      get(const tuple<Types...>&) noexcept;
  template <class T, class... Types>
    constexpr T& get(tuple<Types...>& t) noexcept;
  template <class T, class... Types>
    constexpr T&& get(tuple<Types...>&& t) noexcept;
  template <class T, class... Types>
    constexpr const T& get(const tuple<Types...>& t) noexcept;

  // 20.61.0.1, relational operators:
  template<class... TTypes, class... UTypes>
    constexpr bool operator==(const tuple<TTypes...>&, const tuple<UTypes...>&);
  template<class... TTypes, class... UTypes>
    constexpr bool operator<(const tuple<TTypes...>&, const tuple<UTypes...>&);
  template<class... TTypes, class... UTypes>
    constexpr bool operator!=(const tuple<TTypes...>&, const tuple<UTypes...>&);
  template<class... TTypes, class... UTypes>
    constexpr bool operator>(const tuple<TTypes...>&, const tuple<UTypes...>&);
  template<class... TTypes, class... UTypes>
    constexpr bool operator<=(const tuple<TTypes...>&, const tuple<UTypes...>&);
  template<class... TTypes, class... UTypes>
    constexpr bool operator>=(const tuple<TTypes...>&, const tuple<UTypes...>&);

  // 20.64.0.1, allocator-related traits
  template <class... Types, class Alloc>
    struct uses_allocator<tuple<Types...>, Alloc>;

  // 20.65.0.1, specialized algorithms:
  template <class... Types>
    void swap(tuple<Types...>& x, tuple<Types...>& y) noexcept(see below);
}
```

20.37.1 Class template tuple [tuple.tuple]

```
namespace std {
  template <class... Types>
  class tuple  {
  public:

    // 20.37.1.1, tuple construction
    constexpr tuple();
    constexpr explicit tuple(const Types&...);
    template <class... UTypes>
      constexpr explicit tuple(UTypes&&...);

    tuple(const tuple&) = default;
    tuple(tuple&&) = default;
```

```
    template <class... UTypes>
      constexpr tuple(const tuple<UTypes...>&);
    template <class... UTypes>
      constexpr tuple(tuple<UTypes...>&&);

    template <class U1, class U2>
      constexpr tuple(const pair<U1, U2>&);       // only if sizeof...(Types) == 2
    template <class U1, class U2>
      constexpr tuple(pair<U1, U2>&&);            // only if sizeof...(Types) == 2

    // allocator-extended constructors
    template <class Alloc>
      tuple(allocator_arg_t, const Alloc& a);
    template <class Alloc>
      tuple(allocator_arg_t, const Alloc& a, const Types&...);
    template <class Alloc, class... UTypes>
      tuple(allocator_arg_t, const Alloc& a, UTypes&&...);
    template <class Alloc>
      tuple(allocator_arg_t, const Alloc& a, const tuple&);
    template <class Alloc>
      tuple(allocator_arg_t, const Alloc& a, tuple&&);
    template <class Alloc, class... UTypes>
      tuple(allocator_arg_t, const Alloc& a, const tuple<UTypes...>&);
    template <class Alloc, class... UTypes>
      tuple(allocator_arg_t, const Alloc& a, tuple<UTypes...>&&);
    template <class Alloc, class U1, class U2>
      tuple(allocator_arg_t, const Alloc& a, const pair<U1, U2>&);
    template <class Alloc, class U1, class U2>
      tuple(allocator_arg_t, const Alloc& a, pair<U1, U2>&&);

    // 20.51.0.1, tuple assignment
    tuple& operator=(const tuple&);
    tuple& operator=(tuple&&) noexcept(see below);

    template <class... UTypes>
      tuple& operator=(const tuple<UTypes...>&);
    template <class... UTypes>
      tuple& operator=(tuple<UTypes...>&&);

    template <class U1, class U2>
      tuple& operator=(const pair<U1, U2>&);     // only if sizeof...(Types) == 2
    template <class U1, class U2>
      tuple& operator=(pair<U1, U2>&&);          // only if sizeof...(Types) == 2

    // 20.53.0.1, tuple swap
    void swap(tuple&) noexcept(see below);
  };
}
```

20.37.1.1 Construction [tuple.cnstr]

1 For each `tuple` constructor, an exception is thrown only if the construction of one of the types in `Types` throws an exception.

2

The defaulted move and copy constructor, respectively, of `tuple` shall be a `constexpr` function if and only if all required element-wise initializations for copy and move, respectively, would satisfy the requirements for a `constexpr` function. The defaulted move and copy constructor of `tuple<>` shall be `constexpr` functions.

1 In the constructor descriptions that follow, let i be in the range `[0,sizeof...(Types))` in order, T_i be the ith type in `Types`, and U_i be the ith type in a template parameter pack named `UTypes`, where indexing is zero-based.

```
constexpr tuple();
```

2 *Requires:* `is_default_constructible<`T_i`>::value` is true for all i.

3 *Effects:* Value initializes each element.

```
constexpr explicit tuple(const Types&...);
```

4 *Requires:* `is_copy_constructible<`T_i`>::value` is true for all i.

5 *Effects:* Initializes each element with the value of the corresponding parameter.

```
template <class... UTypes>
  constexpr explicit tuple(UTypes&&... u);
```

1 *Requires:* `sizeof...(Types) == sizeof...(UTypes)`. `is_constructible<`T_i`, `U_i`&&>::value` is `true` for all i.

1 *Effects:* Initializes the elements in the tuple with the corresponding value in `std::forward<UTypes>(u)`.

2 *Remark:* This constructor shall not participate in overload resolution unless each type in `UTypes` is implicitly convertible to its corresponding type in `Types`.

```
tuple(const tuple& u) = default;
```

1 *Requires:* `is_copy_constructible<`T_i`>::value` is `true` for all i.

2 *Effects:* Initializes each element of `*this` with the corresponding element of `u`.

```
tuple(tuple&& u) = default;
```

3 *Requires:* `is_move_constructible<`T_i`>::value` is `true` for all i.

4 *Effects:* For all i, initializes the ith element of `*this` with `std::forward<`T_i`>(get<`i`>(u))`.

```
template <class... UTypes> constexpr tuple(const tuple<UTypes...>& u);
```

1 *Requires:* `sizeof...(Types) == sizeof...(UTypes)`. `is_constructible<`T_i`, const `U_i`&>::value` is `true` for all i.

1 *Effects:* Constructs each element of `*this` with the corresponding element of `u`.

1 *Remark:* This constructor shall not participate in overload resolution unless `const` U_i`&` is implicitly convertible to T_i for all i.

```
template <class... UTypes> constexpr tuple(tuple<UTypes...>&& u);
```

1 *Requires:* `sizeof...(Types) == sizeof...(UTypes)`. `is_constructible<`T_i`, `U_i`&&>::value` is `true` for all i.

2 *Effects:* For all i, initializes the ith element of `*this` with `std::forward<`U_i`>(get<`i`>(u))`.

3 *Remark:* This constructor shall not participate in overload resolution unless each type in `UTypes` is implicitly convertible to its corresponding type in `Types`.

```
template <class U1, class U2> constexpr tuple(const pair<U1, U2>& u);
```

4 *Requires:* `sizeof...(Types) == 2`. `is_constructible<`T_0`, const U1&>::value` is `true` for the first type T_0 in `Types` and `is_constructible<`T_1`, const U2&>::value` is `true` for the second type T_1 in `Types`.

5 *Effects:* Constructs the first element with `u.first` and the second element with `u.second`.

6 *Remark:* This constructor shall not participate in overload resolution unless `const U1&` is implicitly convertible to T_0 and `const U2&` is implicitly convertible to T_1.

```
template <class U1, class U2> constexpr tuple(pair<U1, U2>&& u);
```

7 *Requires:* `sizeof...(Types) == 2`. `is_constructible<`T_0`, U1&&>::value` is `true` for the first type T_0 in `Types` and `is_constructible<`T_1`, U2&&>::value` is `true` for the second type T_1 in `Types`.

8 *Effects:* Initializes the first element with `std::forward<U1>(u.first)` and the second element with `std::forward<U2>(u.second)`.

9 *Remark:* This constructor shall not participate in overload resolution unless `U1` is implicitly convertible to T_0 and `U2` is implicitly convertible to T_1.

```
template <class Alloc>
  tuple(allocator_arg_t, const Alloc& a);
template <class Alloc>
  tuple(allocator_arg_t, const Alloc& a, const Types&...);
template <class Alloc, class... UTypes>
  tuple(allocator_arg_t, const Alloc& a, UTypes&&...);
template <class Alloc>
  tuple(allocator_arg_t, const Alloc& a, const tuple&);
template <class Alloc>
  tuple(allocator_arg_t, const Alloc& a, tuple&&);
template <class Alloc, class... UTypes>
  tuple(allocator_arg_t, const Alloc& a, const tuple<UTypes...>&);
template <class Alloc, class... UTypes>
  tuple(allocator_arg_t, const Alloc& a, tuple<UTypes...>&&);
template <class Alloc, class U1, class U2>
  tuple(allocator_arg_t, const Alloc& a, const pair<U1, U2>&);
template <class Alloc, class U1, class U2>
  tuple(allocator_arg_t, const Alloc& a, pair<U1, U2>&&);
```

10 *Requires:* `Alloc` shall meet the requirements for an `Allocator` (17.165.0.1).

11 *Effects:* Equivalent to the preceding constructors except that each element is constructed with uses-allocator construction (20.86.2.2).

20.51.0.1 Assignment [tuple.assign]

1 For each `tuple` assignment operator, an exception is thrown only if the assignment of one of the types in `Types` throws an exception. In the function descriptions that follow, let i be in the range `[0,sizeof...(Types))` in order, T_i be the ith type in `Types`, and U_i be the ith type in a template parameter pack named `UTypes`, where indexing is zero-based.

```
tuple& operator=(const tuple& u);
```

2 *Requires:* `is_copy_assignable<`T_i`>::value` is `true` for all i.

3 *Effects:* Assigns each element of `u` to the corresponding element of `*this`.

1 *Returns:* `*this`

```
tuple& operator=(tuple&& u) noexcept(see below);
```

2 *Remark:* The expression inside `noexcept` is equivalent to the logical AND of the following expressions:

```
is_nothrow_move_assignable<Ti>::value
```

where T_i is the ith type in `Types`.

3 *Requires:* `is_move_assignable<`T_i`>::value` is `true` for all i.

4 *Effects:* For all i, assigns `std::forward<`T_i`>(get<`i`>(u))` to `get<`i`>(*this)`.

5 *Returns:* `*this`.

```
template <class... UTypes>
  tuple& operator=(const tuple<UTypes...>& u);
```

1 *Requires:* `sizeof...(Types) == sizeof...(UTypes)` and `is_assignable<`T_i`&, const `U_i`&>::value` is `true` for all i.

2 *Effects:* Assigns each element of `u` to the corresponding element of `*this`.

3 *Returns:* `*this`

```
template <class... UTypes>
  tuple& operator=(tuple<UTypes...>&& u);
```

4 *Requires:* `is_assignable<Ti&, Ui&&>::value == true` for all `i`. `sizeof...(Types) == sizeof...(UTypes)`.

5 *Effects:* For all i, assigns `std::forward<`U_i`>(get<`i`>(u))` to `get<`i`>(*this)`.

6 *Returns:* `*this`.

```
template <class U1, class U2> tuple& operator=(const pair<U1, U2>& u);
```

7 *Requires:* `sizeof...(Types) == 2`. `is_assignable<`T_0`&, const U1&>::value` is `true` for the first type T_0 in `Types` and `is_assignable<`T_1`&, const U2&>::value` is `true` for the second type T_1 in `Types`.

8 *Effects:* Assigns `u.first` to the first element of `*this` and `u.second` to the second element of `*this`.

9 *Returns:* `*this`

```
template <class U1, class U2> tuple& operator=(pair<U1, U2>&& u);
```

10 *Requires:* `sizeof...(Types) == 2`. `is_assignable<`T_0`&, U1&&>::value` is `true` for the first type T_0 in `Types` and `is_assignable<`T_1`&, U2&&>::value` is `true` for the second type T_1 in `Types`.

11 *Effects:* Assigns `std::forward<U1>(u.first)` to the first element of `*this` and `std::forward<U2>(u.second)` to the second element of `*this`.

12 *Returns:* `*this`.

20.53.0.1 `swap` [tuple.swap]

```
void swap(tuple& rhs) noexcept(see below);
```

1 *Remark:* The expression inside `noexcept` is equivalent to the logical AND of the following expressions:

```
noexcept(swap(declval<Ti&>>(), declval<Ti&>()))
```

where T_i is the ith type in `Types`.

2 *Requires:* Each element in `*this` shall be swappable with (17.143.0.1) the corresponding element in `rhs`.

3 *Effects:* Calls `swap` for each element in `*this` and its corresponding element in `rhs`.

4 *Throws:* Nothing unless one of the element-wise `swap` calls throws an exception.

20.53.0.2 Tuple creation functions [tuple.creation]

1 In the function descriptions that follow, let i be in the range `[0,sizeof...(TTypes))` in order and let T_i be the ith type in a template parameter pack named `TTypes`; let j be in the range `[0,sizeof...(UTypes))` in order and U_j be the jth type in a template parameter pack named `UTypes`, where indexing is zero-based.

```
template<class... Types>
  constexpr tuple<VTypes...> make_tuple(Types&&... t);
```

2 Let U_i be `decay_t<`T_i`>` for each T_i in `Types`. Then each V_i in `VTypes` is `X&` if U_i equals `reference_wrapper<X>`, otherwise V_i is U_i.

1 *Returns:* `tuple<VTypes...>(std::forward<Types>(t)...)`.

2 [*Example:*

```
int i; float j;
make_tuple(1, ref(i), cref(j))
```

creates a tuple of type

```
tuple<int, int&, const float&>
```

— *end example*]

```
template<class... Types>
  constexpr tuple<Types&&...> forward_as_tuple(Types&&... t) noexcept;
```

3 *Effects:* Constructs a tuple of references to the arguments in `t` suitable for forwarding as arguments to a function. Because the result may contain references to temporary variables, a program shall ensure that the return value of this function does not outlive any of its arguments. (e.g., the program should typically not store the result in a named variable).

4 *Returns:* `tuple<Types&&...>(std::forward<Types>(t)...)`

```
template<class... Types>
  constexpr tuple<Types&...> tie(Types&... t) noexcept;
```

5

Returns: `tuple<Types&...>(t...)`. When an argument in `t` is `ignore`, assigning any value to the corresponding tuple element has no effect.

1 [*Example:* `tie` functions allow one to create tuples that unpack tuples into variables. `ignore` can be used for elements that are not needed:

```
int i; std::string s;
tie(i, ignore, s) = make_tuple(42, 3.14, "C++");
// i == 42, s == "C++"
```

— *end example*]

```
template <class... Tuples>
  constexpr tuple<CTypes...> tuple_cat(Tuples&&... tpls);
```

2 In the following paragraphs, let T_i be the ith type in `Tuples`, U_i be `remove_reference_t<Ti>`, and tp_i be the ith parameter in the function parameter pack `tpls`, where all indexing is zero-based.

3 *Requires:* For all i, U_i shall be the type cv_i `tuple<`$Args_i$`...>`, where cv_i is the (possibly empty) i^{th} cv-qualifier-seq and $Args_i$ is the parameter pack representing the element types in U_i. Let A_{ik} be the ${k_i}^{th}$ type in $Args_i$. For all A_{ik} the following requirements shall be satisfied: If T_i is deduced as an lvalue reference type, then `is_constructible<`A_{ik}`,` cv_i A_{ik}`&>::value == true`, otherwise `is_constructible<`A_{ik}`,` $cv_i A_{ik}$`&&>::value == true`.

4 *Remarks:* The types in `CTypes` shall be equal to the ordered sequence of the extended types $Args_0$`...`, $Args_1$`...`, ... $Args_{n-1}$`...`, where n is equal to `sizeof...(Tuples)`. Let e_i`...` be the ith ordered sequence of tuple elements of the resulting `tuple` object corresponding to the type sequence $Args_i$.

5 *Returns:* A `tuple` object constructed by initializing the ${k_i}^{th}$ type element e_{ik} in e_i`...` with `get<`k_i`>(std::forward<`T_i`>(`tp_i`))` for each valid k_i and each group e_i in order.

6 *Note:* An implementation may support additional types in the parameter pack `Tuples` that support the `tuple`-like protocol, such as `pair` and `array`.

20.56.0.1 Tuple helper classes [tuple.helper]

```
template <class T> struct tuple_size;
```

Remarks: All specializations of `tuple_size<T>` shall meet the `UnaryTypeTrait` requirements (20.450.1) with a `BaseCharacteristic` of `integral_constant<size_t, N>` for some `N`.

```
template <class... Types>
class tuple_size<tuple<Types...> >
  : public integral_constant<size_t, sizeof...(Types)> { };

template <size_t I, class... Types>
class tuple_element<I, tuple<Types...> > {
public:
  typedef TI type;
};
```

1 *Requires:* `I < sizeof...(Types)`. The program is ill-formed if `I` is out of bounds.

2 *Type:* `TI` is the type of the `I`th element of `Types`, where indexing is zero-based.

```
template <class T> class tuple_size<const T>;
template <class T> class tuple_size<volatile T>;
template <class T> class tuple_size<const volatile T>;
```

3

Let *TS* denote `tuple_size<T>` of the *cv*-unqualified type `T`. Then each of the three templates shall meet the `UnaryTypeTrait` requirements (20.450.1) with a `BaseCharacteristic` of

```
integral_constant<size_t, TS::value>
```

```
template <size_t I, class T> class tuple_element<I, const T>;
template <size_t I, class T> class tuple_element<I, volatile T>;
template <size_t I, class T> class tuple_element<I, const volatile T>;
```

Let *TE* denote `tuple_element<I, T>` of the *cv*-unqualified type `T`. Then each of the three templates shall meet the `TransformationTrait` requirements (20.450.1) with a member typedef `type` that names the following type:

— for the first specialization, `add_const_t<`*TE*`::type>`,

— for the second specialization, `add_volatile_t<`*TE*`::type>`, and

— for the third specialization, `add_cv_t<`*TE*`::type>`.

20.57.0.1 Element access [tuple.elem]

```
template <size_t I, class... Types>
  constexpr tuple_element_t<I, tuple<Types...> >& get(tuple<Types...>& t) noexcept;
```

1 *Requires:* `I < sizeof...(Types)`. The program is ill-formed if `I` is out of bounds.

2 *Returns:* A reference to the `I`th element of `t`, where indexing is zero-based.

```
template <size_t I, class... Types>
  constexpr tuple_element_t<I, tuple<Types...> >&& get(tuple<Types...>&& t) noexcept;
```

3 *Effects:* Equivalent to `return std::forward<typename tuple_element<I, tuple<Types...> > ::type&&>(get<I>(t));`

4 *Note:* if a `T` in `Types` is some reference type `X&`, the return type is `X&`, not `X&&`. However, if the element type is a non-reference type `T`, the return type is `T&&`.

```
template <size_t I, class... Types>
  constexpr tuple_element_t<I, tuple<Types...> > const& get(const tuple<Types...>& t) noexcept;
```

5 *Requires:* `I < sizeof...(Types)`. The program is ill-formed if `I` is out of bounds.

6 *Returns:* A const reference to the `I`th element of `t`, where indexing is zero-based.

7 [*Note:* Constness is shallow. If a `T` in `Types` is some reference type `X&`, the return type is `X&`, not `const X&`. However, if the element type is non-reference type `T`, the return type is `const T&`. This is consistent with how constness is defined to work for member variables of reference type. — *end note*]

```
template <class T, class... Types>
  constexpr T& get(tuple<Types...>& t) noexcept;
template <class T, class... Types>
  constexpr T&& get(tuple<Types...>&& t) noexcept;
template <class T, class... Types>
  constexpr const T& get(const tuple<Types...>& t) noexcept;
```

8

Requires: The type `T` occurs exactly once in `Types...`. Otherwise, the program is ill-formed.

1 *Returns:* A reference to the element of `t` corresponding to the type `T` in `Types...`.

2 [*Example:*

```
const tuple<int, const int, double, double> t(1, 2, 3.4, 5.6);
const int &i1 = get<int>(t);        // OK. Not ambiguous. i1 == 1
const int &i2 = get<const int>(t);  // OK. Not ambiguous. i2 == 2
const double &d = get<double>(t);   // ERROR. ill-formed
```

— *end example*]

3

[*Note:* The reason `get` is a nonmember function is that if this functionality had been provided as a member function, code where the type depended on a template parameter would have required using the `template` keyword. — *end note*]

20.61.0.1 Relational operators [tuple.rel]

```
template<class... TTypes, class... UTypes>
  constexpr bool operator==(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```

1

Requires: For all `i`, where `0 <= i` and `i < sizeof...(TTypes)`, `get<i>(t) == get<i>(u)` is a valid expression returning a type that is convertible to `bool`. `sizeof...(TTypes) == sizeof...(UTypes)`.

1 *Returns:* `true` if `get<i>(t) == get<i>(u)` for all `i`, otherwise `false`. For any two zero-length tuples `e` and `f`, `e == f` returns `true`.

2 *Effects:* The elementary comparisons are performed in order from the zeroth index upwards. No comparisons or element accesses are performed after the first equality comparison that evaluates to `false`.

```
template<class... TTypes, class... UTypes>
  constexpr bool operator<(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```

3 *Requires:* For all `i`, where `0 <= i` and `i < sizeof...(TTypes)`, `get<i>(t) < get<i>(u)` and `get<i>(u) < get<i>(t)` are valid expressions returning types that are convertible to `bool`. `sizeof...(TTypes) == sizeof...(UTypes)`.

4 *Returns:* The result of a lexicographical comparison between `t` and `u`. The result is defined as: `(bool)(get<0>(t) < get<0>(u)) || (!(bool)(get<0>(u) < get<0>(t)) && t`$_{\text{tail}}$ `<` `u`$_{\text{tail}}$`)`, where `r`$_{\text{tail}}$ for some tuple `r` is a tuple containing all but the first element of `r`. For any two zero-length tuples `e` and `f`, `e < f` returns `false`.

```
template<class... TTypes, class... UTypes>
  constexpr bool operator!=(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```

1 *Returns:* `!(t == u)`.

```
template<class... TTypes, class... UTypes>
  constexpr bool operator>(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```

2 *Returns:* `u < t`.

```
template<class... TTypes, class... UTypes>
  constexpr bool operator<=(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```

3 *Returns:* `!(u < t)`

```
template<class... TTypes, class... UTypes>
  constexpr bool operator>=(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```

4 *Returns:* `!(t < u)`

5 [*Note:* The above definitions for comparison operators do not require $\texttt{t}_{\text{tail}}$ (or $\texttt{u}_{\text{tail}}$) to be constructed. It may not even be possible, as `t` and `u` are not required to be copy constructible. Also, all comparison operators are short circuited; they do not perform element accesses beyond what is required to determine the result of the comparison. — *end note*]

20.64.0.1 Tuple traits [tuple.traits]

```
template <class... Types, class Alloc>
  struct uses_allocator<tuple<Types...>, Alloc> : true_type { };
```

Requires: `Alloc` shall be an `Allocator` (17.165.0.1).

1

[*Note:* Specialization of this trait informs other library components that `tuple` can be constructed with an allocator, even though it does not have a nested `allocator_type`. — *end note*]

20.65.0.1 Tuple specialized algorithms [tuple.special]

```
template <class... Types>
  void swap(tuple<Types...>& x, tuple<Types...>& y) noexcept(see below);
```

1 *Remark:* The expression inside `noexcept` is equivalent to:

```
noexcept(x.swap(y))
```

2 *Effects:* `x.swap(y)`

20.66 Compile-time integer sequences [intseq]

20.66.1 In general [intseq.general]

1 The library provides a class template that can represent an integer sequence. When used as an argument to a function template the parameter pack defining the sequence can be deduced and used in a pack expansion.

2 [*Example:*

```
template<class F, class Tuple, std::size_t... I>
  decltype(auto) apply_impl(F&& f, Tuple&& t, index_sequence<I...>) {
    return std::forward<F>(f)(std::get<I>(std::forward<Tuple>(t))...);
  }

template<class F, class Tuple>
  decltype(auto) apply(F&& f, Tuple&& t) {
    using Indices = make_index_sequence<std::tuple_size<std::decay_t<Tuple>>::value>;
    return apply_impl(std::forward<F>(f), std::forward<Tuple>(t), Indices());
  }
```

— *end example*] [*Note:* The `index_sequence` alias template is provided for the common case of an integer sequence of type `size_t`. — *end note*]

20.66.2 Class template `integer_sequence` [intseq.intseq]

```
namespace std {
  template<class T, T... I>
  struct integer_sequence {
    typedef T value_type;
    static constexpr size_t size() noexcept { return sizeof...(I); }
  };
}
```

1 T shall be an integer type.

20.66.3 Alias template make_integer_sequence [intseq.make]

```
template<class T, T N>
  using make_integer_sequence = integer_sequence<T, see below>;
```

1

If N is negative the program is ill-formed. The alias template make_integer_sequence denotes a specialization of integer_sequence with N template non-type arguments. The type make_integer_sequence<T, N> denotes the type integer_sequence<T, 0, 1, ..., N-1>. [*Note:* make_integer_sequence<int, 0> denotes the type integer_sequence<int> — *end note*]

20.68 Class template bitset [template.bitset]

Header <bitset> synopsis

```
#include <string>
#include <iosfwd>                // for istream, ostream
namespace std {
  template <size_t N> class bitset;

  // 20.78.2 bitset operators:
  template <size_t N>
    bitset<N> operator&(const bitset<N>&, const bitset<N>&) noexcept;
  template <size_t N>
    bitset<N> operator|(const bitset<N>&, const bitset<N>&) noexcept;
  template <size_t N>
    bitset<N> operator^(const bitset<N>&, const bitset<N>&) noexcept;
  template <class charT, class traits, size_t N>
    basic_istream<charT, traits>&
    operator>>(basic_istream<charT, traits>& is, bitset<N>& x);
  template <class charT, class traits, size_t N>
    basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const bitset<N>& x);
}
```

1 The header <bitset> defines a class template and several related functions for representing and manipulating fixed-size sequences of bits.

```
namespace std {
  template<size_t N> class bitset {
  public:
    // bit reference:
    class reference {
      friend class bitset;
      reference() noexcept;
    public:
     ~reference() noexcept;
      reference& operator=(bool x) noexcept;              // for b[i] = x;
      reference& operator=(const reference&) noexcept;    // for b[i] = b[j];
      bool operator~() const noexcept;                    // flips the bit
      operator bool() const noexcept;                     // for x = b[i];
      reference& flip() noexcept;                         // for b[i].flip();
    };

    // 20.70.1 constructors:
```

```
  constexpr bitset() noexcept;
  constexpr bitset(unsigned long long val) noexcept;
  template<class charT, class traits, class Allocator>
    explicit bitset(
      const basic_string<charT,traits,Allocator>& str,
      typename basic_string<charT,traits,Allocator>::size_type pos = 0,
      typename basic_string<charT,traits,Allocator>::size_type n =
        basic_string<charT,traits,Allocator>::npos,
        charT zero = charT('0'), charT one = charT('1'));
  template <class charT>
    explicit bitset(
      const charT* str,
      typename basic_string<charT>::size_type n = basic_string<charT>::npos,
      charT zero = charT('0'), charT one = charT('1'));

  // 20.75.1 bitset operations:
  bitset<N>& operator&=(const bitset<N>& rhs) noexcept;
  bitset<N>& operator|=(const bitset<N>& rhs) noexcept;
  bitset<N>& operator^=(const bitset<N>& rhs) noexcept;
  bitset<N>& operator<<=(size_t pos) noexcept;
  bitset<N>& operator>>=(size_t pos) noexcept;
  bitset<N>& set() noexcept;
  bitset<N>& set(size_t pos, bool val = true);
  bitset<N>& reset() noexcept;
  bitset<N>& reset(size_t pos);
  bitset<N>  operator~() const noexcept;
  bitset<N>& flip() noexcept;
  bitset<N>& flip(size_t pos);

  // element access:
  constexpr bool operator[](size_t pos) const;       // for b[i];
  reference operator[](size_t pos);                  // for b[i];

  unsigned long to_ulong() const;
  unsigned long long to_ullong() const;
  template <class charT = char,
      class traits = char_traits<charT>,
      class Allocator = allocator<charT> >
    basic_string<charT, traits, Allocator>
    to_string(charT zero = charT('0'), charT one = charT('1')) const;
  size_t count() const noexcept;
  constexpr size_t size() const noexcept;
  bool operator==(const bitset<N>& rhs) const noexcept;
  bool operator!=(const bitset<N>& rhs) const noexcept;
  bool test(size_t pos) const;
  bool all() const noexcept;
  bool any() const noexcept;
  bool none() const noexcept;
  bitset<N> operator<<(size_t pos) const noexcept;
  bitset<N> operator>>(size_t pos) const noexcept;
};

// 20.78.1 hash support
template <class T> struct hash;
template <size_t N> struct hash<bitset<N> >;
```

```
}
```

2 The class template `bitset<N>`describes an object that can store a sequence consisting of a fixed number of bits, `N`.

3

Each bit represents either the value zero (reset) or one (set). To *toggle* a bit is to change the value zero to one, or the value one to zero. Each bit has a non-negative position `pos`. When converting between an object of class `bitset<N>` and a value of some integral type, bit position `pos` corresponds to the *bit value* `1 <<pos`. The integral value corresponding to two or more bits is the sum of their bit values.

1 The functions described in this subclause can report three kinds of errors, each associated with a distinct exception:

(1.1) — an *invalid-argument* error is associated with exceptions of type `invalid_argument` (19.3.3);

(1.2) — an *out-of-range* error is associated with exceptions of type `out_of_range` (19.4.1);

(1.3) — an *overflow* error is associated with exceptions of type `overflow_error` (19.7.1).

20.70.1 `bitset` constructors [bitset.cons]

```
constexpr bitset() noexcept;
```

1

Effects: Constructs an object of class `bitset<N>`, initializing all bits to zero.

```
constexpr bitset(unsigned long long val) noexcept;
```

1

Effects: Constructs an object of class `bitset<N>`, initializing the first `M` bit positions to the corresponding bit values in `val`. `M` is the smaller of `N` and the number of bits in the value representation (3.2955) of `unsigned long long`. If `M < N`, the remaining bit positions are initialized to zero.

```
template <class charT, class traits, class Allocator>
explicit
bitset(const basic_string<charT, traits, Allocator>& str,
       typename basic_string<charT, traits, Allocator>::size_type pos = 0,
       typename basic_string<charT, traits, Allocator>::size_type n =
         basic_string<charT, traits, Allocator>::npos,
         charT zero = charT('0'), charT one = charT('1'));
```

1 *Requires:* `pos <= str.size()`.

2 *Throws:* `out_of_range` if `pos > str.size()`.

3 *Effects:* Determines the effective length `rlen` of the initializing string as the smaller of `n` and `str.size() - pos`.

The function then throws `invalid_argument` if any of the `rlen` characters in `str` beginning at position `pos` is other than `zero` or `one`. The function uses `traits::eq()` to compare the character values.

Otherwise, the function constructs an object of class `bitset<N>`, initializing the first `M` bit positions to values determined from the corresponding characters in the string `str`. `M` is the smaller of `N` and `rlen`.

4 An element of the constructed object has value zero if the corresponding character in `str`, beginning at position `pos`, is `zero`. Otherwise, the element has the value one. Character position `pos + M - 1` corresponds to bit position zero. Subsequent decreasing character positions correspond to increasing bit positions.

5 If `M < N`, remaining bit positions are initialized to zero.

```
template <class charT>
  explicit bitset(
    const charT* str,
    typename basic_string<charT>::size_type n = basic_string<charT>::npos,
    charT zero = charT('0'), charT one = charT('1'));
```

6 *Effects:* Constructs an object of class `bitset<N>` as if by

```
bitset(
  n == basic_string<charT>::npos
    ? basic_string<charT>(str)
    : basic_string<charT>(str, n),
  0, n, zero, one)
```

20.75.1 `bitset` members [bitset.members]

`bitset<N>& operator&=(const bitset<N>& rhs) noexcept;`

1 *Effects:* Clears each bit in `*this` for which the corresponding bit in `rhs` is clear, and leaves all other bits unchanged.

2 *Returns:* `*this`.

`bitset<N>& operator|=(const bitset<N>& rhs) noexcept;`

3 *Effects:* Sets each bit in `*this` for which the corresponding bit in `rhs` is set, and leaves all other bits unchanged.

4 *Returns:* `*this`.

`bitset<N>& operator^=(const bitset<N>& rhs) noexcept;`

5 *Effects:* Toggles each bit in `*this` for which the corresponding bit in `rhs` is set, and leaves all other bits unchanged.

6 *Returns:* `*this`.

`bitset<N>& operator<<=(size_t pos) noexcept;`

7 *Effects:* Replaces each bit at position `I` in `*this` with a value determined as follows:

(7.1) — If `I < pos`, the new value is zero;

(7.2) — If `I >= pos`, the new value is the previous value of the bit at position `I - pos`.

8 *Returns:* `*this`.

`bitset<N>& operator>>=(size_t pos) noexcept;`

9 *Effects:* Replaces each bit at position `I` in `*this` with a value determined as follows:

(9.1) — If `pos >= N - I`, the new value is zero;

(9.2) — If `pos < N - I`, the new value is the previous value of the bit at position `I + pos`.

10 *Returns:* `*this`.

`bitset<N>& set() noexcept;`

11 *Effects:* Sets all bits in `*this`.

12 *Returns:* `*this`.

`bitset<N>& set(size_t pos, bool val = true);`

13 *Requires:* `pos` is valid

14 *Throws:* `out_of_range` if `pos` does not correspond to a valid bit position.

15 *Effects:* Stores a new value in the bit at position `pos` in `*this`. If `val` is nonzero, the stored value is one, otherwise it is zero.

16 *Returns:* `*this`.

```
bitset<N>& reset() noexcept;
```

17 *Effects:* Resets all bits in `*this`.

18 *Returns:* `*this`.

```
bitset<N>& reset(size_t pos);
```

19 *Requires:* `pos` is valid

20 *Throws:* `out_of_range` if `pos` does not correspond to a valid bit position.

21 *Effects:* Resets the bit at position `pos` in `*this`.

22 *Returns:* `*this`.

```
bitset<N> operator~() const noexcept;
```

23 *Effects:* Constructs an object `x` of class `bitset<N>` and initializes it with `*this`.

24 *Returns:* `x.flip()`.

```
bitset<N>& flip() noexcept;
```

25 *Effects:* Toggles all bits in `*this`.

26 *Returns:* `*this`.

```
bitset<N>& flip(size_t pos);
```

27 *Requires:* `pos` is valid

28 *Throws:* `out_of_range` if `pos` does not correspond to a valid bit position.

29 *Effects:* Toggles the bit at position `pos` in `*this`.

30 *Returns:* `*this`.

```
unsigned long to_ulong() const;
```

31 *Throws:* `overflow_error` if the integral value `x` corresponding to the bits in `*this` cannot be represented as type `unsigned long`.

32 *Returns:* `x`.

```
unsigned long long to_ullong() const;
```

33 *Throws:* `overflow_error` if the integral value `x` corresponding to the bits in `*this` cannot be represented as type `unsigned long long`.

34 *Returns:* `x`.

```
template <class charT = char,
    class traits = char_traits<charT>,
    class Allocator = allocator<charT> >
  basic_string<charT, traits, Allocator>
  to_string(charT zero = charT('0'), charT one = charT('1')) const;
```

35

Effects: Constructs a string object of the appropriate type and initializes it to a string of length `N` characters. Each character is determined by the value of its corresponding bit position in `*this`. Character position `N - 1` corresponds to bit position zero. Subsequent decreasing character positions correspond to increasing bit positions. Bit value zero becomes the character `zero`, bit value one becomes the character `one`.

1 *Returns:* The created object.

```
size_t count() const noexcept;
```

2 *Returns:* A count of the number of bits set in `*this`.

```
constexpr size_t size() const noexcept;
```

3 *Returns:* `N`.

```
bool operator==(const bitset<N>& rhs) const noexcept;
```

4 *Returns:* `true` if the value of each bit in `*this` equals the value of the corresponding bit in `rhs`.

```
bool operator!=(const bitset<N>& rhs) const noexcept;
```

5 *Returns:* `true` if `!(*this == rhs)`.

```
bool test(size_t pos) const;
```

6 *Requires:* `pos` is valid

7 *Throws:* `out_of_range` if `pos` does not correspond to a valid bit position.

8 *Returns:* `true` if the bit at position `pos` in `*this` has the value one.

```
bool all() const noexcept;
```

9 *Returns:* `count() == size()`

```
bool any() const noexcept;
```

10 *Returns:* `count() != 0`

```
bool none() const noexcept;
```

11 *Returns:* `count() == 0`

```
bitset<N> operator<<(size_t pos) const noexcept;
```

12 *Returns:* `bitset<N>(*this) <<= pos`.

```
bitset<N> operator>>(size_t pos) const noexcept;
```

13 *Returns:* `bitset<N>(*this) >>= pos`.

```
constexpr bool operator[](size_t pos) const;
```

14 *Requires:* `pos` shall be valid.

15 *Returns:* `true` if the bit at position `pos` in `*this` has the value one, otherwise `false`.

16 *Throws:* Nothing.

```
bitset<N>::reference operator[](size_t pos);
```

17 *Requires:* `pos` shall be valid.

18

Returns: An object of type `bitset<N>::reference` such that `(*this)[pos] == this->test(pos)`, and such that `(*this)[pos] = val` is equivalent to `this->set(pos, val)`.

1 *Throws:* Nothing.

2

Remark: For the purpose of determining the presence of a data race (1.859), any access or update through the resulting reference potentially accesses or modifies, respectively, the entire underlying bitset.

20.78.1 `bitset` hash support [bitset.hash]

```
template <size_t N> struct hash<bitset<N> >;
```

1 The template specialization shall meet the requirements of class template `hash` (20.443.1).

20.78.2 `bitset` operators [bitset.operators]

```
bitset<N> operator&(const bitset<N>& lhs, const bitset<N>& rhs) noexcept;
```

1 *Returns:* `bitset<N>(lhs) &= rhs`.

```
bitset<N> operator|(const bitset<N>& lhs, const bitset<N>& rhs) noexcept;
```

2 *Returns:* `bitset<N>(lhs) |= rhs`.

```
bitset<N> operator^(const bitset<N>& lhs, const bitset<N>& rhs) noexcept;
```

3 *Returns:* `bitset<N>(lhs) ^= rhs`.

```
template <class charT, class traits, size_t N>
  basic_istream<charT, traits>&
  operator>>(basic_istream<charT, traits>& is, bitset<N>& x);
```

4 A formatted input function (27.128.0.1).

5

Effects: Extracts up to `N` characters from `is`. Stores these characters in a temporary object `str` of type `basic_string<charT, traits>`, then evaluates the expression `x = bitset<N>(str)`. Characters are extracted and stored until any of the following occurs:

— `N` characters have been extracted and stored;

— end-of-file occurs on the input sequence;

— the next input character is neither `is.widen('0')` nor `is.widen('1')` (in which case the input character is not extracted).

1

If no characters are stored in `str`, calls `is.setstate(ios_base::failbit)` (which may throw `ios_base::failure` (27.84.0.1)).

1

Returns: `is`.

```
template <class charT, class traits, size_t N>
  basic_ostream<charT, traits>&
  operator<<(basic_ostream<charT, traits>& os, const bitset<N>& x);
```

1 *Returns:*

```
os << x.template to_string<charT,traits,allocator<charT> >(
  use_facet<ctype<charT> >(os.getloc()).widen('0'),
  use_facet<ctype<charT> >(os.getloc()).widen('1'))
```

(see 27.198.0.1).

20.82 Memory [memory]

20.82.1 In general [memory.general]

1 This subclause describes the contents of the header `<memory>` (20.82.2) and some of the contents of the C headers `<cstdlib>` and `<cstring>` (20.116.1).

20.82.2 Header `<memory>` synopsis [memory.syn]

1 The header `<memory>` defines several types and function templates that describe properties of pointers and pointer-like types, manage memory for containers and other template types, and construct multiple objects in uninitialized memory buffers (20.82.3–20.115.1). The header also defines the templates `unique_ptr`, `shared_ptr`, `weak_ptr`, and various function templates that operate on objects of these types (20.127).

```
namespace std {
  // 20.82.3, pointer traits
  template <class Ptr> struct pointer_traits;
  template <class T> struct pointer_traits<T*>;

  // 20.85.1, pointer safety
  enum class pointer_safety { relaxed, preferred, strict };
  void declare_reachable(void* p);
  template <class T> T* undeclare_reachable(T* p);
  void declare_no_pointers(char* p, size_t n);
  void undeclare_no_pointers(char* p, size_t n);
  pointer_safety get_pointer_safety() noexcept;

  // 20.85.2, pointer alignment function
  void* align(std::size_t alignment, std::size_t size,
    void*& ptr, std::size_t& space);

  // 20.86.1, allocator argument tag
  struct allocator_arg_t { };
  constexpr allocator_arg_t allocator_arg{};

  // 20.86.2, uses_allocator
  template <class T, class Alloc> struct uses_allocator;

  // 20.86.3, allocator traits
  template <class Alloc> struct allocator_traits;

  // 20.92.1, the default allocator:
  template <class T> class allocator;
  template <> class allocator<void>;
  template <class T, class U>
    bool operator==(const allocator<T>&, const allocator<U>&) noexcept;
  template <class T, class U>
    bool operator!=(const allocator<T>&, const allocator<U>&) noexcept;

  // 20.113.1, raw storage iterator:
```

```
template <class OutputIterator, class T> class raw_storage_iterator;

// 20.114.1, temporary buffers:
template <class T>
  pair<T*,ptrdiff_t> get_temporary_buffer(ptrdiff_t n) noexcept;
template <class T>
  void return_temporary_buffer(T* p);

// 20.115.1, specialized algorithms:
template <class T> T* addressof(T& r) noexcept;
template <class InputIterator, class ForwardIterator>
  ForwardIterator uninitialized_copy(InputIterator first, InputIterator last,
                                     ForwardIterator result);
template <class InputIterator, class Size, class ForwardIterator>
  ForwardIterator uninitialized_copy_n(InputIterator first, Size n,
                                       ForwardIterator result);
template <class ForwardIterator, class T>
  void uninitialized_fill(ForwardIterator first, ForwardIterator last,
                          const T& x);
template <class ForwardIterator, class Size, class T>
  ForwardIterator uninitialized_fill_n(ForwardIterator first, Size n, const T& x);

// 20.127.1 class template unique_ptr:
template <class T> struct default_delete;
template <class T> struct default_delete<T[]>;
template <class T, class D = default_delete<T>> class unique_ptr;
template <class T, class D> class unique_ptr<T[], D>;

template <class T, class... Args> unique_ptr<T> make_unique(Args&&... args);
template <class T> unique_ptr<T> make_unique(size_t n);
template <class T, class... Args> unspecified  make_unique(Args&&...) = delete;

template <class T, class D> void swap(unique_ptr<T, D>& x, unique_ptr<T, D>& y) noexcept;

template <class T1, class D1, class T2, class D2>
  bool operator==(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
template <class T1, class D1, class T2, class D2>
  bool operator!=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
template <class T1, class D1, class T2, class D2>
  bool operator<(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
template <class T1, class D1, class T2, class D2>
  bool operator<=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
template <class T1, class D1, class T2, class D2>
  bool operator>(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
template <class T1, class D1, class T2, class D2>
  bool operator>=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);

template <class T, class D>
  bool operator==(const unique_ptr<T, D>& x, nullptr_t) noexcept;
template <class T, class D>
  bool operator==(nullptr_t, const unique_ptr<T, D>& y) noexcept;
template <class T, class D>
  bool operator!=(const unique_ptr<T, D>& x, nullptr_t) noexcept;
template <class T, class D>
  bool operator!=(nullptr_t, const unique_ptr<T, D>& y) noexcept;
```

```
template <class T, class D>
  bool operator<(const unique_ptr<T, D>& x, nullptr_t);
template <class T, class D>
  bool operator<(nullptr_t, const unique_ptr<T, D>& y);
template <class T, class D>
  bool operator<=(const unique_ptr<T, D>& x, nullptr_t);
template <class T, class D>
  bool operator<=(nullptr_t, const unique_ptr<T, D>& y);
template <class T, class D>
  bool operator>(const unique_ptr<T, D>& x, nullptr_t);
template <class T, class D>
  bool operator>(nullptr_t, const unique_ptr<T, D>& y);
template <class T, class D>
  bool operator>=(const unique_ptr<T, D>& x, nullptr_t);
template <class T, class D>
  bool operator>=(nullptr_t, const unique_ptr<T, D>& y);

// 20.189.1.1, class bad_weak_ptr:
class bad_weak_ptr;

// 20.189.1.2, class template shared_ptr:
template<class T> class shared_ptr;

// 20.247.0.0.1, shared_ptr creation
template<class T, class... Args> shared_ptr<T> make_shared(Args&&... args);
template<class T, class A, class... Args>
  shared_ptr<T> allocate_shared(const A& a, Args&&... args);

// 20.262.0.0.1, shared_ptr comparisons:
template<class T, class U>
  bool operator==(shared_ptr<T> const& a, shared_ptr<U> const& b) noexcept;
template<class T, class U>
  bool operator!=(shared_ptr<T> const& a, shared_ptr<U> const& b) noexcept;
template<class T, class U>
  bool operator<(shared_ptr<T> const& a, shared_ptr<U> const& b) noexcept;
template<class T, class U>
  bool operator>(shared_ptr<T> const& a, shared_ptr<U> const& b) noexcept;
template<class T, class U>
  bool operator<=(shared_ptr<T> const& a, shared_ptr<U> const& b) noexcept;
template<class T, class U>
  bool operator>=(shared_ptr<T> const& a, shared_ptr<U> const& b) noexcept;

template <class T>
  bool operator==(const shared_ptr<T>& x, nullptr_t) noexcept;
template <class T>
  bool operator==(nullptr_t, const shared_ptr<T>& y) noexcept;
template <class T>
  bool operator!=(const shared_ptr<T>& x, nullptr_t) noexcept;
template <class T>
  bool operator!=(nullptr_t, const shared_ptr<T>& y) noexcept;
template <class T>
  bool operator<(const shared_ptr<T>& x, nullptr_t) noexcept;
template <class T>
  bool operator<(nullptr_t, const shared_ptr<T>& y) noexcept;
template <class T>
```

```
  bool operator<=(const shared_ptr<T>& x, nullptr_t) noexcept;
template <class T>
  bool operator<=(nullptr_t, const shared_ptr<T>& y) noexcept;
template <class T>
  bool operator>(const shared_ptr<T>& x, nullptr_t) noexcept;
template <class T>
  bool operator>(nullptr_t, const shared_ptr<T>& y) noexcept;
template <class T>
  bool operator>=(const shared_ptr<T>& x, nullptr_t) noexcept;
template <class T>
  bool operator>=(nullptr_t, const shared_ptr<T>& y) noexcept;

// 20.262.0.0.2, shared_ptr specialized algorithms:
template<class T> void swap(shared_ptr<T>& a, shared_ptr<T>& b) noexcept;

// 20.262.0.0.3, shared_ptr casts:
template<class T, class U>
  shared_ptr<T> static_pointer_cast(shared_ptr<U> const& r) noexcept;
template<class T, class U>
  shared_ptr<T> dynamic_pointer_cast(shared_ptr<U> const& r) noexcept;
template<class T, class U>
  shared_ptr<T> const_pointer_cast(shared_ptr<U> const& r) noexcept;

// 20.263.0.0.1, shared_ptr get_deleter:
template<class D, class T> D* get_deleter(shared_ptr<T> const& p) noexcept;

// 20.264.0.0.1, shared_ptr I/O:
template<class E, class T, class Y>
  basic_ostream<E, T>& operator<< (basic_ostream<E, T>& os, shared_ptr<Y> const& p);

// 20.264.0.1, class template weak_ptr:
template<class T> class weak_ptr;

// 20.266.0.0.1, weak_ptr specialized algorithms:
template<class T> void swap(weak_ptr<T>& a, weak_ptr<T>& b) noexcept;

// 20.266.0.1, class template owner_less:
template<class T> class owner_less;

// 20.266.0.2, class template enable_shared_from_this:
template<class T> class enable_shared_from_this;

// 20.279.0.1, shared_ptr atomic access:
template<class T>
  bool atomic_is_lock_free(const shared_ptr<T>* p);

template<class T>
  shared_ptr<T> atomic_load(const shared_ptr<T>* p);
template<class T>
  shared_ptr<T> atomic_load_explicit(const shared_ptr<T>* p, memory_order mo);

template<class T>
  void atomic_store(shared_ptr<T>* p, shared_ptr<T> r);
template<class T>
  void atomic_store_explicit(shared_ptr<T>* p, shared_ptr<T> r, memory_order mo);
```

```
  template<class T>
    shared_ptr<T> atomic_exchange(shared_ptr<T>* p, shared_ptr<T> r);
  template<class T>
    shared_ptr<T> atomic_exchange_explicit(shared_ptr<T>* p, shared_ptr<T> r,
                                           memory_order mo);

  template<class T>
    bool atomic_compare_exchange_weak(
      shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w);
  template<class T>
    bool atomic_compare_exchange_strong(
      shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w);
  template<class T>
    bool atomic_compare_exchange_weak_explicit(
      shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w,
      memory_order success, memory_order failure);
  template<class T>
    bool atomic_compare_exchange_strong_explicit(
      shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w,
      memory_order success, memory_order failure);

  // 20.282.0.1 hash support
  template <class T> struct hash;
  template <class T, class D> struct hash<unique_ptr<T, D> >;
  template <class T> struct hash<shared_ptr<T> >;

  // D.59, auto_ptr (deprecated)
  template <class X> class auto_ptr;
}
```

20.82.3 Pointer traits [pointer.traits]

1 The class template `pointer_traits` supplies a uniform interface to certain attributes of pointer-like types.

```
namespace std {
  template <class Ptr> struct pointer_traits {
    typedef Ptr       pointer;
    typedef see below element_type;
    typedef see below difference_type;

    template <class U> using rebind = see below;

    static pointer pointer_to(see below r);
  };

  template <class T> struct pointer_traits<T*> {
    typedef T*        pointer;
    typedef T         element_type;
    typedef ptrdiff_t difference_type;

    template <class U> using rebind = U*;

    static pointer pointer_to(see below r) noexcept;
  };
}
```

20.82.3.1 Pointer traits member types [pointer.traits.types]

`typedef see below element_type;`

1 *Type:* `Ptr::element_type` if such a type exists; otherwise, `T` if `Ptr` is a class template instantiation of the form `SomePointer<T, Args>`, where `Args` is zero or more type arguments; otherwise, the specialization is ill-formed.

`typedef see below difference_type;`

1 *Type:* `Ptr::difference_type` if such a type exists; otherwise, `std::ptrdiff_t`.

`template <class U> using rebind = see below;`

2 *Alias template:* `Ptr::rebind<U>` if such a type exists; otherwise, `SomePointer<U, Args>` if `Ptr` is a class template instantiation of the form `SomePointer<T, Args>`, where `Args` is zero or more type arguments; otherwise, the instantiation of `rebind` is ill-formed.

20.85.0.1 Pointer traits member functions [pointer.traits.functions]

```
static pointer pointer_traits::pointer_to(see below r);
static pointer pointer_traits<T*>::pointer_to(see below r) noexcept;
```

1 *Remark:* If `element_type` is (possibly cv-qualified) `void`, the type of `r` is unspecified; otherwise, it is `element_type&`.

2 *Returns:* The first member function returns a pointer to `r` obtained by calling `Ptr::pointer_to(r)` through which indirection is valid; an instantiation of this function is ill-formed if `Ptr` does not have a matching `pointer_to` static member function. The second member function returns `std::addressof(r)`.

20.85.1 Pointer safety [util.dynamic.safety]

1 A complete object is *declared reachable* while the number of calls to `declare_reachable` with an argument referencing the object exceeds the number of calls to `undeclare_reachable` with an argument referencing the object.

`void declare_reachable(void* p);`

2 *Requires:* `p` shall be a safely-derived pointer (3.2532.0.1) or a null pointer value.

3 *Effects:* If `p` is not null, the complete object referenced by `p` is subsequently declared reachable (3.2532.0.1).

4 *Throws:* May throw `std::bad_alloc` if the system cannot allocate additional memory that may be required to track objects declared reachable.

`template <class T> T* undeclare_reachable(T* p);`

5 *Requires:* If `p` is not null, the complete object referenced by `p` shall have been previously declared reachable, and shall be live (3.2563) from the time of the call until the last `undeclare_reachable(p)` call on the object.

6 *Returns:* A safely derived copy of `p` which shall compare equal to `p`.

7 *Throws:* Nothing.

8 [*Note:* It is expected that calls to `declare_reachable(p)` will consume a small amount of memory in addition to that occupied by the referenced object until the matching call to `undeclare_reachable(p)` is encountered. Long running programs should arrange that calls are matched. — *end note*]

`void declare_no_pointers(char* p, size_t n);`

9 *Requires:* No bytes in the specified range are currently registered with `declare_no_pointers()`. If the specified range is in an allocated object, then it must be entirely within a single allocated object. The object must be live until the corresponding `undeclare_no_pointers()` call. [*Note:* In a garbage-collecting implementation, the fact that a region in an object is registered with `declare_no_pointers()` should not prevent the object from being collected. — *end note*]

10 *Effects:* The `n` bytes starting at `p` no longer contain traceable pointer locations, independent of their type. Hence indirection through a pointer located there is undefined if the object it points to was created by global `operator new` and not previously declared reachable. [*Note:* This may be used to inform a garbage collector or leak detector that this region of memory need not be traced. — *end note*]

11 *Throws:* Nothing.

12 [*Note:* Under some conditions implementations may need to allocate memory. However, the request can be ignored if memory allocation fails. — *end note*]

```
void undeclare_no_pointers(char* p, size_t n);
```

13 *Requires:* The same range must previously have been passed to `declare_no_pointers()`.

14 *Effects:* Unregisters a range registered with `declare_no_pointers()` for destruction. It must be called before the lifetime of the object ends.

15 *Throws:* Nothing.

```
pointer_safety get_pointer_safety() noexcept;
```

16 *Returns:* `pointer_safety::strict` if the implementation has strict pointer safety (3.2532.0.1). It is implementation defined whether `get_pointer_safety` returns `pointer_safety::relaxed` or `pointer_safety::preferred` if the implementation has relaxed pointer safety.[229]

20.85.2 Align [ptr.align]

```
void* align(std::size_t alignment, std::size_t size,
    void*& ptr, std::size_t& space);
```

1 *Effects:* If it is possible to fit `size` bytes of storage aligned by `alignment` into the buffer pointed to by `ptr` with length `space`, the function updates `ptr` to point to the first possible address of such storage and decreases `space` by the number of bytes used for alignment. Otherwise, the function does nothing.

2 *Requires:*

— `alignment` shall be a fundamental alignment value or an extended alignment value supported by the implementation in this context

— `ptr` shall point to contiguous storage of at least `space` bytes

1 *Returns:* A null pointer if the requested aligned buffer would not fit into the available space, otherwise the adjusted value of `ptr`.

2 [*Note:* The function updates its `ptr` and `space` arguments so that it can be called repeatedly with possibly different `alignment` and `size` arguments for the same buffer. — *end note*]

229) `pointer_safety::preferred` might be returned to indicate that a leak detector is running so that the program can avoid spurious leak reports.

20.86.1 Allocator argument tag [allocator.tag]

```
namespace std {
  struct allocator_arg_t { };
  constexpr allocator_arg_t allocator_arg{};
}
```

1 The `allocator_arg_t` struct is an empty structure type used as a unique type to disambiguate constructor and function overloading. Specifically, several types (see `tuple` 20.34) have constructors with `allocator_arg_t` as the first argument, immediately followed by an argument of a type that satisfies the `Allocator` requirements (17.165.0.1).

20.86.2 `uses_allocator` [allocator.uses]

20.86.2.1 `uses_allocator` trait [allocator.uses.trait]

```
template <class T, class Alloc> struct uses_allocator;
```

1 *Remark:* automatically detects whether `T` has a nested `allocator_type` that is convertible from `Alloc`. Meets the BinaryTypeTrait requirements (20.450.1). The implementation shall provide a definition that is derived from `true_type` if a type `T::allocator_type` exists and `is_convertible<Alloc, T::allocator_type>::value != false`, otherwise it shall be derived from `false_type`. A program may specialize this template to derive from `true_type` for a user-defined type `T` that does not have a nested `allocator_type` but nonetheless can be constructed with an allocator where either:

(1.1) — the first argument of a constructor has type `allocator_arg_t` and the second argument has type `Alloc` or

(1.2) — the last argument of a constructor has type `Alloc`.

20.86.2.2 uses-allocator construction [allocator.uses.construction]

1 *Uses-allocator construction* with allocator `Alloc` refers to the construction of an object `obj` of type `T`, using constructor arguments `v1, v2, ..., vN` of types `V1, V2, ..., VN`, respectively, and an allocator `alloc` of type `Alloc`, according to the following rules:

(1.1) — if `uses_allocator<T, Alloc>::value` is `false` and `is_constructible<T, V1, V2, ..., VN>::value` is `true`, then `obj` is initialized as `obj(v1, v2, ..., vN)`;

(1.2) — otherwise, if `uses_allocator<T, Alloc>::value` is `true` and `is_constructible<T, allocator_arg_t, Alloc, V1, V2, ..., VN>::value` is `true`, then `obj` is initialized as `obj(allocator_arg, alloc, v1, v2, ..., vN)`;

(1.3) — otherwise, if `uses_allocator<T, Alloc>::value` is `true` and `is_constructible<T, V1, V2, ..., VN, Alloc>::value` is `true`, then `obj` is initialized as `obj(v1, v2, ..., vN, alloc)`;

(1.4) — otherwise, the request for uses-allocator construction is ill-formed. [*Note:* An error will result if `uses_allocator<T, Alloc>::value` is `true` but the specific constructor does not take an allocator. This definition prevents a silent failure to pass the allocator to an element. — *end note*]

20.86.3 Allocator traits [allocator.traits]

1 The class template `allocator_traits` supplies a uniform interface to all allocator types. An allocator cannot be a non-class type, however, even if `allocator_traits` supplies the entire required interface. [*Note:* Thus, it is always possible to create a derived class from an allocator. — *end note*]

```
namespace std {
  template <class Alloc> struct allocator_traits {
    typedef Alloc allocator_type;
```

```
    typedef typename Alloc::value_type value_type;

    typedef see below pointer;
    typedef see below const_pointer;
    typedef see below void_pointer;
    typedef see below const_void_pointer;

    typedef see below difference_type;
    typedef see below size_type;

    typedef see below propagate_on_container_copy_assignment;
    typedef see below propagate_on_container_move_assignment;
    typedef see below propagate_on_container_swap;

    template <class T> using rebind_alloc = see below;
    template <class T> using rebind_traits = allocator_traits<rebind_alloc<T> >;

    static pointer allocate(Alloc& a, size_type n);
    static pointer allocate(Alloc& a, size_type n, const_void_pointer hint);

    static void deallocate(Alloc& a, pointer p, size_type n);

    template <class T, class... Args>
      static void construct(Alloc& a, T* p, Args&&... args);

    template <class T>
      static void destroy(Alloc& a, T* p);

    static size_type max_size(const Alloc& a) noexcept;

    static Alloc select_on_container_copy_construction(const Alloc& rhs);
  };
}
```

20.86.3.1 Allocator traits member types [allocator.traits.types]

`typedef see below pointer;`

1 *Type:* `Alloc::pointer` if such a type exists; otherwise, `value_type*`.

`typedef see below const_pointer;`

2 *Type:* `Alloc::const_pointer` if such a type exists; otherwise, `pointer_traits<pointer>::rebind<const value_type>`.

`typedef see below void_pointer;`

3 *Type:* `Alloc::void_pointer` if such a type exists; otherwise, `pointer_traits<pointer>::rebind<void>`.

`typedef see below const_void_pointer;`

4 *Type:* `Alloc::const_void_pointer` if such a type exists; otherwise, `pointer_traits<pointer>::rebind<const void>`.

`typedef see below difference_type;`

5 *Type:* `Alloc::difference_type` if such a type exists; otherwise, `pointer_traits<pointer>::difference_type`.

```
typedef see below size_type;
```

6 *Type:* `Alloc::size_type` if such a type exists; otherwise, `make_unsigned_t<difference_type>`.

```
typedef see below propagate_on_container_copy_assignment;
```

7 *Type:* `Alloc::propagate_on_container_copy_assignment` if such a type exists, otherwise `false_type`.

```
typedef see below propagate_on_container_move_assignment;
```

8 *Type:* `Alloc::propagate_on_container_move_assignment` if such a type exists, otherwise `false_type`.

```
typedef see below propagate_on_container_swap;
```

1 *Type:* `Alloc::propagate_on_container_swap` if such a type exists, otherwise `false_type`.

```
template <class T> using rebind_alloc = see below;
```

2 *Alias template:* `Alloc::rebind<T>::other` if such a type exists; otherwise, `Alloc<T, Args>` if `Alloc` is a class template instantiation of the form `Alloc<U, Args>`, where `Args` is zero or more type arguments; otherwise, the instantiation of `rebind_alloc` is ill-formed.

20.89.0.1 Allocator traits static member functions [allocator.traits.members]

```
static pointer allocate(Alloc& a, size_type n);
```

1 *Returns:* `a.allocate(n)`.

```
static pointer allocate(Alloc& a, size_type n, const_void_pointer hint);
```

2 *Returns:* `a.allocate(n, hint)` if that expression is well-formed; otherwise, `a.allocate(n)`.

```
static void deallocate(Alloc& a, pointer p, size_type n);
```

3 *Effects:* calls `a.deallocate(p, n)`.

4 *Throws:* Nothing.

```
template <class T, class... Args>
  static void construct(Alloc& a, T* p, Args&&... args);
```

5 *Effects:* calls `a.construct(p, std::forward<Args>(args)...)` if that call is well-formed; otherwise, invokes `::new (static_cast<void*>(p)) T(std::forward<Args>(args)...)`.

```
template <class T>
  static void destroy(Alloc& a, T* p);
```

1 *Effects:* calls `a.destroy(p)` if that call is well-formed; otherwise, invokes `p->~T()`.

```
static size_type max_size(const Alloc& a) noexcept;
```

2 *Returns:* `a.max_size()` if that expression is well-formed; otherwise, `numeric_limits<size_type>::max()`.

```
static Alloc select_on_container_copy_construction(const Alloc& rhs);
```

3 *Returns:* `rhs.select_on_container_copy_construction()` if that expression is well-formed; otherwise, `rhs`.

20.92.1 The default allocator [default.allocator]

```
namespace std {
  template <class T> class allocator;

  // specialize for void:
  template <> class allocator<void> {
  public:
    typedef void*   pointer;
    typedef const void* const_pointer;
    // reference-to-void members are impossible.
    typedef void  value_type;
    template <class U> struct rebind { typedef allocator<U> other; };
  };

  template <class T> class allocator {
   public:
    typedef size_t    size_type;
    typedef ptrdiff_t difference_type;
    typedef T*        pointer;
    typedef const T*  const_pointer;
    typedef T&        reference;
    typedef const T&  const_reference;
    typedef T         value_type;
    template <class U> struct rebind { typedef allocator<U> other; };
    typedef true_type propagate_on_container_move_assignment;

    allocator() noexcept;
    allocator(const allocator&) noexcept;
    template <class U> allocator(const allocator<U>&) noexcept;
   ~allocator();

    pointer address(reference x) const noexcept;
    const_pointer address(const_reference x) const noexcept;

    pointer allocate(
      size_type, allocator<void>::const_pointer hint = 0);
    void deallocate(pointer p, size_type n);
    size_type max_size() const noexcept;

    template<class U, class... Args>
      void construct(U* p, Args&&... args);
    template <class U>
      void destroy(U* p);
  };
}
```

20.92.1.1 `allocator` members [allocator.members]

1

Except for the destructor, member functions of the default allocator shall not introduce data races (1.859) as a result of concurrent calls to those member functions from different threads. Calls to these functions that allocate or deallocate a particular unit of storage shall occur in a single total order, and each such deallocation call shall happen before the next allocation (if any) in this order.

```
pointer address(reference x) const noexcept;
```

1 *Returns:* The actual address of the object referenced by `x`, even in the presence of an overloaded operator&.

```
const_pointer address(const_reference x) const noexcept;
```

2

Returns: The actual address of the object referenced by `x`, even in the presence of an overloaded operator&.

```
pointer allocate(size_type n, allocator<void>::const_pointer hint = 0);
```

1 [*Note:* In a container member function, the address of an adjacent element is often a good choice to pass for the `hint` argument. — *end note*]

2

Returns: A pointer to the initial element of an array of storage of size `n * sizeof(T)`, aligned appropriately for objects of type `T`. It is implementation-defined whether over-aligned types are supported (3.3924).

1

Remark: the storage is obtained by calling `::operator new(std::size_t)` (18.159.1), but it is unspecified when or how often this function is called. The use of `hint` is unspecified, but intended as an aid to locality if an implementation so desires.

1 *Throws:* `bad_alloc` if the storage cannot be obtained.

```
void deallocate(pointer p, size_type n);
```

2

Requires: `p` shall be a pointer value obtained from `allocate()`. `n` shall equal the value passed as the first argument to the invocation of allocate which returned `p`.

1 *Effects:* Deallocates the storage referenced by `p` .

2

Remarks: Uses `::operator delete(void*, std::size_t)` (18.159.1), but it is unspecified when this function is called.

```
size_type max_size() const noexcept;
```

1

Returns: The largest value N for which the call `allocate(N,0)` might succeed.

```
template <class U, class... Args>
  void construct(U* p, Args&&... args);
```

1 *Effects:* `::new((void *)p) U(std::forward<Args>(args)...)`

```
template <class U>
  void destroy(U* p);
```

2 *Effects:* `p->~U()`

20.113.0.1 `allocator` globals [allocator.globals]

```
template <class T1, class T2>
  bool operator==(const allocator<T1>&, const allocator<T2>&) noexcept;
```

1 *Returns:* `true`.

```
template <class T1, class T2>
  bool operator!=(const allocator<T1>&, const allocator<T2>&) noexcept;
```

2 *Returns:* `false`.

20.113.1 Raw storage iterator [storage.iterator]

1

`raw_storage_iterator` is provided to enable algorithms to store their results into uninitialized memory. The template parameter `OutputIterator` is required to have its `operator*` return an object for which `operator&` is defined and returns a pointer to `T`, and is also required to satisfy the requirements of an output iterator (24.116.1).

```
namespace std {
  template <class OutputIterator, class T>
  class raw_storage_iterator
    : public iterator<output_iterator_tag,void,void,void,void> {
  public:
    explicit raw_storage_iterator(OutputIterator x);

    raw_storage_iterator& operator*();
    raw_storage_iterator& operator=(const T& element);
    raw_storage_iterator& operator++();
    raw_storage_iterator  operator++(int);
  };
}
```

`explicit raw_storage_iterator(OutputIterator x);`

1 *Effects:* Initializes the iterator to point to the same value to which `x` points.

`raw_storage_iterator& operator*();`

2 *Returns:* `*this`

`raw_storage_iterator& operator=(const T& element);`

3 *Effects:* Constructs a value from `element` at the location to which the iterator points.

4 *Returns:* A reference to the iterator.

`raw_storage_iterator& operator++();`

5 *Effects:* Pre-increment: advances the iterator and returns a reference to the updated iterator.

`raw_storage_iterator operator++(int);`

6 *Effects:* Post-increment: advances the iterator and returns the old value of the iterator.

20.114.1 Temporary buffers [temporary.buffer]

```
template <class T>
  pair<T*, ptrdiff_t> get_temporary_buffer(ptrdiff_t n) noexcept;
```

1

Effects: Obtains a pointer to storage sufficient to store up to `n` adjacent `T` objects. It is implementation-defined whether over-aligned types are supported (3.3924).

1 *Returns:* A `pair` containing the buffer's address and capacity (in the units of `sizeof(T)`), or a pair of 0 values if no storage can be obtained or if `n <= 0`.

```
template <class T> void return_temporary_buffer(T* p);
```

2 *Effects:* Deallocates the buffer to which `p` points.

3 *Requires:* The buffer shall have been previously allocated by `get_temporary_buffer`.

20.115.1 Specialized algorithms [specialized.algorithms]

1

All the iterators that are used as template parameters in the following algorithms are required to have their `operator*` return an object for which `operator&` is defined and returns a pointer to `T`. In the algorithm `uninitialized_copy`, the template parameter `InputIterator` is required to satisfy the requirements of an input iterator (24.100.1). In all of the following algorithms, the template parameter `ForwardIterator` is required to satisfy the requirements of a forward iterator (24.122.1), and is required to have the property that no exceptions are thrown from increment, assignment, comparison, or indirection through valid iterators. In the following algorithms, if an exception is thrown there are no effects.

20.116.0.1 `addressof` [specialized.addressof]

```
template <class T> T* addressof(T& r) noexcept;
```

1 *Returns:* The actual address of the object or function referenced by `r`, even in the presence of an overloaded `operator&`.

20.116.0.2 `uninitialized_copy` [uninitialized.copy]

```
template <class InputIterator, class ForwardIterator>
  ForwardIterator uninitialized_copy(InputIterator first, InputIterator last,
                                     ForwardIterator result);
```

1 *Effects:*

```
for (; first != last; ++result, ++first)
  ::new (static_cast<void*>(&*result))
    typename iterator_traits<ForwardIterator>::value_type(*first);
```

2 *Returns:* `result`

```
template <class InputIterator, class Size, class ForwardIterator>
  ForwardIterator uninitialized_copy_n(InputIterator first, Size n,
                                       ForwardIterator result);
```

3 *Effects:*

```
for ( ; n > 0; ++result, ++first, --n) {
  ::new (static_cast<void*>(&*result))
    typename iterator_traits<ForwardIterator>::value_type(*first);
}
```

4 *Returns:* `result`

20.116.0.3 uninitialized_fill [uninitialized.fill]

```
template <class ForwardIterator, class T>
  void uninitialized_fill(ForwardIterator first, ForwardIterator last,
                          const T& x);
```

1 *Effects:*

```
for (; first != last; ++first)
  ::new (static_cast<void*>(&*first))
    typename iterator_traits<ForwardIterator>::value_type(x);
```

20.116.0.4 uninitialized_fill_n [uninitialized.fill.n]

```
template <class ForwardIterator, class Size, class T>
  ForwardIterator uninitialized_fill_n(ForwardIterator first, Size n, const T& x);
```

1 *Effects:*

```
for (; n--; ++first)
  ::new (static_cast<void*>(&*first))
    typename iterator_traits<ForwardIterator>::value_type(x);
return first;
```

20.116.1 C library [c.malloc]

1 Table 45 describes the header `<cstdlib>`.

Table 45 — Header `<cstdlib>` synopsis

Type	Name(s)	
Functions:	`calloc`	`malloc`
	`free`	`realloc`

2

The contents are the same as the Standard C library header `<stdlib.h>`, with the following changes:

1

The functions `calloc()`, `malloc()`, and `realloc()` do not attempt to allocate storage by calling `::operator new()` (18.159).

1

The function `free()` does not attempt to deallocate storage by calling `::operator delete()`.

See also: ISO C Clause 7.11.2.

1

Storage allocated directly with `malloc()`, `calloc()`, or `realloc()` is implicitly declared reachable (see 3.2532.0.1) on allocation, ceases to be declared reachable on deallocation, and need not cease to be declared reachable as the result of an `undeclare_reachable()` call. [*Note:* This allows existing C libraries to remain unaffected by restrictions on pointers that are not safely derived, at the expense of providing far fewer garbage collection and leak detection options for `malloc()`-allocated objects. It also allows `malloc()` to be implemented with a separate allocation arena, bypassing the normal `declare_reachable()` implementation. The above functions should never intentionally be used as a replacement for `declare_reachable()`, and newly written code is strongly encouraged to treat memory allocated with these functions as though it were allocated with `operator new`. — *end note*]

Table 46 — Header `<cstring>` synopsis

Type	Name(s)	
Macro:	`NULL`	
Type:	`size_t`	
Functions:	`memchr`	`memcmp`
`memcpy`	`memmove`	`memset`

1 Table 46 describes the header `<cstring>`.

2

The contents are the same as the Standard C library header `<string.h>`, with the change to `memchr()` specified in 21.261.

See also: ISO C Clause 7.11.2.

20.127 Smart pointers [smartptr]

20.127.1 Class template `unique_ptr` [unique.ptr]

1

A *unique pointer* is an object that owns another object and manages that other object through a pointer. More precisely, a unique pointer is an object *u* that stores a pointer to a second object *p* and will dispose of *p* when *u* is itself destroyed (e.g., when leaving block scope (6.419)). In this context, *u* is said to *own* `p`.

1 The mechanism by which *u* disposes of *p* is known as *p*'s associated *deleter*, a function object whose correct invocation results in *p*'s appropriate disposition (typically its deletion).

2 Let the notation *u.p* denote the pointer stored by *u*, and let *u.d* denote the associated deleter. Upon request, *u* can *reset* (replace) *u.p* and *u.d* with another pointer and deleter, but must properly dispose of its owned object via the associated deleter before such replacement is considered completed.

3

Additionally, *u* can, upon request, *transfer ownership* to another unique pointer *u2*. Upon completion of such a transfer, the following postconditions hold:

— *u2.p* is equal to the pre-transfer *u.p*,

— *u.p* is equal to `nullptr`, and

— if the pre-transfer *u.d* maintained state, such state has been transferred to *u2.d*.

As in the case of a reset, *u2* must properly dispose of its pre-transfer owned object via the pre-transfer associated deleter before the ownership transfer is considered complete. [*Note:* A deleter's state need never be copied, only moved or swapped as ownership is transferred. — *end note*]

1

Each object of a type `U` instantiated from the `unique_ptr` template specified in this subclause has the strict ownership semantics, specified above, of a unique pointer. In partial satisfaction of these semantics, each such `U` is `MoveConstructible` and `MoveAssignable`, but is not `CopyConstructible` nor `CopyAssignable`. The template parameter `T` of `unique_ptr` may be an incomplete type.

1 [*Note:* The uses of `unique_ptr` include providing exception safety for dynamically allocated memory, passing ownership of dynamically allocated memory to a function, and returning dynamically allocated memory from a function. — *end note*]

```
namespace std {
```

```
template<class T> struct default_delete;
template<class T> struct default_delete<T[]>;

template<class T, class D = default_delete<T>> class unique_ptr;
template<class T, class D> class unique_ptr<T[], D>;

template<class T, class... Args> unique_ptr<T> make_unique(Args&&... args);
template<class T> unique_ptr<T> make_unique(size_t n);
template<class T, class... Args> unspecified  make_unique(Args&&...) = delete;

template<class T, class D> void swap(unique_ptr<T, D>& x, unique_ptr<T, D>& y) noexcept;

template<class T1, class D1, class T2, class D2>
  bool operator==(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
template<class T1, class D1, class T2, class D2>
  bool operator!=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
template<class T1, class D1, class T2, class D2>
  bool operator<(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
template<class T1, class D1, class T2, class D2>
  bool operator<=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
template<class T1, class D1, class T2, class D2>
  bool operator>(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
template<class T1, class D1, class T2, class D2>
  bool operator>=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);

template <class T, class D>
  bool operator==(const unique_ptr<T, D>& x, nullptr_t) noexcept;
template <class T, class D>
  bool operator==(nullptr_t, const unique_ptr<T, D>& y) noexcept;
template <class T, class D>
  bool operator!=(const unique_ptr<T, D>& x, nullptr_t) noexcept;
template <class T, class D>
  bool operator!=(nullptr_t, const unique_ptr<T, D>& y) noexcept;
template <class T, class D>
  bool operator<(const unique_ptr<T, D>& x, nullptr_t);
template <class T, class D>
  bool operator<(nullptr_t, const unique_ptr<T, D>& y);
template <class T, class D>
  bool operator<=(const unique_ptr<T, D>& x, nullptr_t);
template <class T, class D>
  bool operator<=(nullptr_t, const unique_ptr<T, D>& y);
template <class T, class D>
  bool operator>(const unique_ptr<T, D>& x, nullptr_t);
template <class T, class D>
  bool operator>(nullptr_t, const unique_ptr<T, D>& y);
template <class T, class D>
  bool operator>=(const unique_ptr<T, D>& x, nullptr_t);
template <class T, class D>
  bool operator>=(nullptr_t, const unique_ptr<T, D>& y);

}
```

20.141.0.1 Default deleters [unique.ptr.dltr]

20.141.0.1.1 In general [unique.ptr.dltr.general]

1

The class template `default_delete` serves as the default deleter (destruction policy) for the class template `unique_ptr`.

1

The template parameter `T` of `default_delete` may be an incomplete type.

20.144.0.0.1 `default_delete` [unique.ptr.dltr.dflt]

```
namespace std {
  template <class T> struct default_delete {
    constexpr default_delete() noexcept = default;
    template <class U> default_delete(const default_delete<U>&) noexcept;
    void operator()(T*) const;
  };
}
```

`template <class U> default_delete(const default_delete<U>& other) noexcept;`

1 *Effects:* Constructs a `default_delete` object from another `default_delete<U>` object.

2 *Remarks:* This constructor shall not participate in overload resolution unless `U*` is implicitly convertible to `T*`.

`void operator()(T* ptr) const;`

3 *Effects:* calls `delete` on `ptr`.

4 *Remarks:* If `T` is an incomplete type, the program is ill-formed.

20.144.0.0.2 `default_delete<T[]>` [unique.ptr.dltr.dflt1]

```
namespace std {
  template <class T> struct default_delete<T[]> {
    constexpr default_delete() noexcept = default;
    void operator()(T*) const;
    template <class U> void operator()(U*) const = delete;
  };
}
```

`void operator()(T* ptr) const;`

1 *Effects:* calls `delete[]` on `ptr`.

2 *Remarks:* If T is an incomplete type, the program is ill-formed.

20.144.0.1 `unique_ptr` for single objects [unique.ptr.single]

```
namespace std {
  template <class T, class D = default_delete<T>> class unique_ptr {
  public:
    typedef see below pointer;
    typedef T element_type;
    typedef D deleter_type;

    // 20.150.0.0.1, constructors
```

```
    constexpr unique_ptr() noexcept;
    explicit unique_ptr(pointer p) noexcept;
    unique_ptr(pointer p, see below d1) noexcept;
    unique_ptr(pointer p, see below d2) noexcept;
    unique_ptr(unique_ptr&& u) noexcept;
    constexpr unique_ptr(nullptr_t) noexcept
      : unique_ptr() { }
    template <class U, class E>
      unique_ptr(unique_ptr<U, E>&& u) noexcept;
    template <class U>
      unique_ptr(auto_ptr<U>&& u) noexcept;

    // 20.157.0.0.1, destructor
    ~unique_ptr();

    // 20.165.0.0.1, assignment
    unique_ptr& operator=(unique_ptr&& u) noexcept;
    template <class U, class E> unique_ptr& operator=(unique_ptr<U, E>&& u) noexcept;
    unique_ptr& operator=(nullptr_t) noexcept;

    // 20.170.0.0.1, observers
    add_lvalue_reference_t<T> operator*() const;
    pointer operator->() const noexcept;
    pointer get() const noexcept;
    deleter_type& get_deleter() noexcept;
    const deleter_type& get_deleter() const noexcept;
    explicit operator bool() const noexcept;

    // 20.171.0.0.1 modifiers
    pointer release() noexcept;
    void reset(pointer p = pointer()) noexcept;
    void swap(unique_ptr& u) noexcept;

    // disable copy from lvalue
    unique_ptr(const unique_ptr&) = delete;
    unique_ptr& operator=(const unique_ptr&) = delete;
  };
}
```

1

The default type for the template parameter `D` is `default_delete`. A client-supplied template argument `D` shall be a function object type (20.283), lvalue-reference to function, or lvalue-reference to function object type for which, given a value `d` of type `D` and a value `ptr` of type `unique_ptr<T, D>::pointer`, the expression `d(ptr)` is valid and has the effect of disposing of the pointer as appropriate for that deleter.

1 If the deleter's type `D` is not a reference type, `D` shall satisfy the requirements of `Destructible` (Table 24).

2

If the type `remove_reference_t<D>::pointer` exists, then `unique_ptr<T, D>::pointer` shall be a synonym for `remove_reference_t<D>::pointer`. Otherwise `unique_ptr<T, D>::pointer` shall be a synonym for `T*`. The type `unique_ptr<T, D>::pointer` shall satisfy the requirements of `NullablePointer` (17.153.0.1).

1 [*Example:* Given an allocator type `X` (17.165.0.1) and letting `A` be a synonym for `allocator_traits<X>`, the types `A::pointer`, `A::const_pointer`, `A::void_pointer`, and `A::const_void_pointer` may be used

as `unique_ptr<T, D>::pointer`. — *end example*]

20.150.0.0.1 `unique_ptr` constructors [unique.ptr.single.ctor]

`constexpr unique_ptr() noexcept;`

1 *Requires:* `D` shall satisfy the requirements of `DefaultConstructible` (Table 19), and that construction shall not throw an exception.

2 *Effects:* Constructs a `unique_ptr` object that owns nothing, value-initializing the stored pointer and the stored deleter.

1 *Postconditions:* `get() == nullptr`. `get_deleter()` returns a reference to the stored deleter.

2 *Remarks:* If this constructor is instantiated with a pointer type or reference type for the template argument `D`, the program is ill-formed.

`explicit unique_ptr(pointer p) noexcept;`

1 *Requires:* `D` shall satisfy the requirements of `DefaultConstructible` (Table 19), and that construction shall not throw an exception.

2 *Effects:* Constructs a `unique_ptr` which owns `p`, initializing the stored pointer with `p` and value-initializing the stored deleter.

3 *Postconditions:* `get() == p`. `get_deleter()` returns a reference to the stored deleter.

4 *Remarks:* If this constructor is instantiated with a pointer type or reference type for the template argument `D`, the program is ill-formed.

```
unique_ptr(pointer p, see below d1) noexcept;
unique_ptr(pointer p, see below d2) noexcept;
```

5 The signature of these constructors depends upon whether `D` is a reference type. If `D` is non-reference type `A`, then the signatures are:

```
unique_ptr(pointer p, const A& d);
unique_ptr(pointer p, A&& d);
```

1 If `D` is an lvalue-reference type `A&`, then the signatures are:

```
unique_ptr(pointer p, A& d);
unique_ptr(pointer p, A&& d);
```

2 If `D` is an lvalue-reference type `const A&`, then the signatures are:

```
unique_ptr(pointer p, const A& d);
unique_ptr(pointer p, const A&& d);
```

3 *Requires:*

(3.1) — If `D` is not an lvalue-reference type then

(3.1.1) — If `d` is an lvalue or `const` rvalue then the first constructor of this pair will be selected. `D` shall satisfy the requirements of `CopyConstructible` (Table 21), and the copy constructor of `D` shall not throw an exception. This `unique_ptr` will hold a copy of `d`.

(3.1.2) — Otherwise, `d` is a non-const rvalue and the second constructor of this pair will be selected. `D` shall satisfy the requirements of `MoveConstructible` (Table 20), and the move constructor of `D` shall not throw an exception. This `unique_ptr` will hold a value move constructed from `d`.

(3.2) — Otherwise `D` is an lvalue-reference type. `d` shall be reference-compatible with one of the constructors. If `d` is an rvalue, it will bind to the second constructor of this pair and the program is ill-formed. [*Note:* The diagnostic could be implemented using a `static_assert` which assures that `D` is not a reference type. — *end note*] Else `d` is an lvalue and will bind to the first constructor of this pair. The type which `D` references need not be `CopyConstructible` nor `MoveConstructible`. This `unique_ptr` will hold a `D` which refers to the lvalue `d`. [*Note:* `D` may not be an rvalue-reference type. — *end note*]

4 *Effects:* Constructs a `unique_ptr` object which owns `p`, initializing the stored pointer with `p` and initializing the deleter as described above.

5 *Postconditions:* `get() == p`. `get_deleter()` returns a reference to the stored deleter. If `D` is a reference type then `get_deleter()` returns a reference to the lvalue `d`.

[*Example:*

```
D d;
unique_ptr<int, D> p1(new int, D());        // D must be MoveConstructible
unique_ptr<int, D> p2(new int, d);          // D must be CopyConstructible
unique_ptr<int, D&> p3(new int, d);         // p3 holds a reference to d
unique_ptr<int, const D&> p4(new int, D()); // error: rvalue deleter object combined
                                            // with reference deleter type
```

— *end example*]

`unique_ptr(unique_ptr&& u) noexcept;`

6 *Requires:* If `D` is not a reference type, `D` shall satisfy the requirements of `MoveConstructible` (Table 20). Construction of the deleter from an rvalue of type `D` shall not throw an exception.

7

Effects: Constructs a `unique_ptr` by transferring ownership from `u` to `*this`. If `D` is a reference type, this deleter is copy constructed from `u`'s deleter; otherwise, this deleter is move constructed from `u`'s deleter. [*Note:* The deleter constructor can be implemented with `std::forward<D>`. — *end note*]

1 *Postconditions:* `get()` yields the value `u.get()` yielded before the construction. `get_deleter()` returns a reference to the stored deleter that was constructed from `u.get_deleter()`. If `D` is a reference type then `get_deleter()` and `u.get_deleter()` both reference the same lvalue deleter.

`template <class U, class E> unique_ptr(unique_ptr<U, E>&& u) noexcept;`

2 *Requires:* If `E` is not a reference type, construction of the deleter from an rvalue of type `E` shall be well formed and shall not throw an exception. Otherwise, `E` is a reference type and construction of the deleter from an lvalue of type `E` shall be well formed and shall not throw an exception.

3 *Remarks:* This constructor shall not participate in overload resolution unless:

(3.1) — `unique_ptr<U, E>::pointer` is implicitly convertible to `pointer`,

(3.2) — `U` is not an array type, and

(3.3) — either `D` is a reference type and `E` is the same type as `D`, or `D` is not a reference type and `E` is implicitly convertible to `D`.

4 *Effects:* Constructs a `unique_ptr` by transferring ownership from `u` to `*this`. If `E` is a reference type, this deleter is copy constructed from `u`'s deleter; otherwise, this deleter is move constructed from `u`'s deleter. [*Note:* The deleter constructor can be implemented with `std::forward<E>`. — *end note*]

5 *Postconditions:* `get()` yields the value `u.get()` yielded before the construction. `get_deleter()` returns a reference to the stored deleter that was constructed from `u.get_deleter()`.

```
template <class U>
  unique_ptr(auto_ptr<U>&& u) noexcept;
```

6 *Effects:* Constructs a `unique_ptr` object, initializing the stored pointer with `u.release()` and value-initializing the stored deleter.

7 *Postconditions:* `get()` yields the value `u.get()` yielded before the construction. `u.get() == nullptr`. `get_deleter()` returns a reference to the stored deleter.

8

Remarks: This constructor shall not participate in overload resolution unless `U*` is implicitly convertible to `T*` and `D` is the same type as `default_delete<T>`.

20.157.0.0.1 `unique_ptr` destructor [unique.ptr.single.dtor]

```
~unique_ptr();
```

1

Requires: The expression `get_deleter()(get())` shall be well formed, shall have well-defined behavior, and shall not throw exceptions. [*Note:* The use of `default_delete` requires `T` to be a complete type. — *end note*]

1

Effects: If `get() == nullptr` there are no effects. Otherwise `get_deleter()(get())`.

20.165.0.0.1 `unique_ptr` assignment [unique.ptr.single.asgn]

```
unique_ptr& operator=(unique_ptr&& u) noexcept;
```

1

Requires: If `D` is not a reference type, `D` shall satisfy the requirements of `MoveAssignable` (Table 22) and assignment of the deleter from an rvalue of type `D` shall not throw an exception. Otherwise, `D` is a reference type; `remove_reference_t<D>` shall satisfy the `CopyAssignable` requirements and assignment of the deleter from an lvalue of type `D` shall not throw an exception.

1

Effects: Transfers ownership from `u` to `*this` as if by calling `reset(u.release())` followed by `get_deleter() = std::forward<D>(u.get_deleter())`.

1 *Returns:* `*this`.

```
template <class U, class E> unique_ptr& operator=(unique_ptr<U, E>&& u) noexcept;
```

2 *Requires:* If `E` is not a reference type, assignment of the deleter from an rvalue of type `E` shall be well-formed and shall not throw an exception. Otherwise, `E` is a reference type and assignment of the deleter from an lvalue of type `E` shall be well-formed and shall not throw an exception.

3 *Remarks:* This operator shall not participate in overload resolution unless:

(3.1) — `unique_ptr<U, E>::pointer` is implicitly convertible to `pointer` and

(3.2) — `U` is not an array type.

4 *Effects:* Transfers ownership from `u` to `*this` as if by calling `reset(u.release())` followed by `get_deleter() = std::forward<E>(u.get_deleter())`.

5 *Returns:* `*this`.

```
unique_ptr& operator=(nullptr_t) noexcept;
```

6 *Effects:* `reset()`.

7 *Postcondition:* `get() == nullptr`

8 *Returns:* `*this`.

20.170.0.0.1 `unique_ptr` observers [unique.ptr.single.observers]

```
add_lvalue_reference_t<T> operator*() const;
```

1 *Requires:* `get() != nullptr`.

2 *Returns:* `*get()`.

```
pointer operator->() const noexcept;
```

3 *Requires:* `get() != nullptr`.

4 *Returns:* `get()`.

5

Note: use typically requires that `T` be a complete type.

```
pointer get() const noexcept;
```

1 *Returns:* The stored pointer.

```
deleter_type& get_deleter() noexcept;
const deleter_type& get_deleter() const noexcept;
```

2 *Returns:* A reference to the stored deleter.

```
explicit operator bool() const noexcept;
```

3 *Returns:* `get() != nullptr`.

20.171.0.0.1 `unique_ptr` modifiers [unique.ptr.single.modifiers]

```
pointer release() noexcept;
```

1 *Postcondition:* `get() == nullptr`.

2

Returns: The value `get()` had at the start of the call to `release`.

```
void reset(pointer p = pointer()) noexcept;
```

1 *Requires:* The expression `get_deleter()(get())` shall be well formed, shall have well-defined behavior, and shall not throw exceptions.

2

Effects: assigns `p` to the stored pointer, and then if the old value of the stored pointer, `old_p`, was not equal to `nullptr`, calls `get_deleter()(old_p)`. [*Note:* The order of these operations is significant because the call to `get_deleter()` may destroy `*this`. — *end note*]

1 *Postconditions:* `get() == p`. [*Note:* The postcondition does not hold if the call to `get_deleter()` destroys `*this` since `this->get()` is no longer a valid expression. — *end note*]

```
void swap(unique_ptr& u) noexcept;
```

2 *Requires:* `get_deleter()` shall be swappable (17.143.0.1) and shall not throw an exception under `swap`.

3 *Effects:* Invokes `swap` on the stored pointers and on the stored deleters of `*this` and `u`.

20.177.0.1 unique_ptr for array objects with a runtime length [unique.ptr.runtime]

```
namespace std {
  template <class T, class D> class unique_ptr<T[], D> {
  public:
    typedef see below pointer;
    typedef T element_type;
    typedef D deleter_type;

    // 20.178.0.0.1, constructors
    constexpr unique_ptr() noexcept;
    explicit unique_ptr(pointer p) noexcept;
    unique_ptr(pointer p, see below d) noexcept;
    unique_ptr(pointer p, see below d) noexcept;
    unique_ptr(unique_ptr&& u) noexcept;
    constexpr unique_ptr(nullptr_t) noexcept : unique_ptr() { }

    // destructor
    ~unique_ptr();

    // assignment
    unique_ptr& operator=(unique_ptr&& u) noexcept;
    unique_ptr& operator=(nullptr_t) noexcept;

    // 20.178.0.0.2, observers
    T& operator[](size_t i) const;
    pointer get() const noexcept;
    deleter_type& get_deleter() noexcept;
    const deleter_type& get_deleter() const noexcept;
    explicit operator bool() const noexcept;

    // 20.178.0.0.3 modifiers
    pointer release() noexcept;
    void reset(pointer p = pointer()) noexcept;
    void reset(nullptr_t) noexcept;
    template <class U> void reset(U) = delete;
    void swap(unique_ptr& u) noexcept;

    // disable copy from lvalue
    unique_ptr(const unique_ptr&) = delete;
    unique_ptr& operator=(const unique_ptr&) = delete;
  };
}
```

1 A specialization for array types is provided with a slightly altered interface.

(1.1) — Conversions between different types of `unique_ptr<T[], D>` or to or from the non-array forms of `unique_ptr` produce an ill-formed program.

(1.2) — Pointers to types derived from `T` are rejected by the constructors, and by `reset`.

(1.3) — The observers `operator*` and `operator->` are not provided.

(1.4) — The indexing observer `operator[]` is provided.

(1.5) — The default deleter will call `delete[]`.

2

Descriptions are provided below only for member functions that have behavior different from the primary template.

1 The template argument `T` shall be a complete type.

20.178.0.0.1 `unique_ptr` constructors [unique.ptr.runtime.ctor]

```
explicit unique_ptr(pointer p) noexcept;
unique_ptr(pointer p, see below d) noexcept;
unique_ptr(pointer p, see below d) noexcept;
```

These constructors behave the same as in the primary template except that they do not accept pointer types which are convertible to `pointer`. [*Note:* One implementation technique is to create private templated overloads of these members. — *end note*]

20.178.0.0.2 `unique_ptr` observers [unique.ptr.runtime.observers]

```
T& operator[](size_t i) const;
```

1 *Requires:* `i <` the number of elements in the array to which the stored pointer points.

2 *Returns:* `get()[i]`.

20.178.0.0.3 `unique_ptr` modifiers [unique.ptr.runtime.modifiers]

```
void reset(nullptr_t p) noexcept;
```

1 *Effects:* Equivalent to `reset(pointer())`.

20.178.0.1 `unique_ptr` creation [unique.ptr.create]

```
template <class T, class... Args> unique_ptr<T> make_unique(Args&&... args);
```

1

Remarks: This function shall not participate in overload resolution unless `T` is not an array.

1

Returns: `unique_ptr<T>(new T(std::forward<Args>(args)...))`.

```
template <class T> unique_ptr<T> make_unique(size_t n);
```

1

Remarks: This function shall not participate in overload resolution unless `T` is an array of unknown bound.

1

Returns: `unique_ptr<T>(new remove_extent_t<T>[n]())`.

```
template <class T, class... Args> unspecified  make_unique(Args&&...) = delete;
```

1

Remarks: This function shall not participate in overload resolution unless `T` is an array of known bound.

20.189.0.1 unique_ptr specialized algorithms [unique.ptr.special]

```
template <class T, class D> void swap(unique_ptr<T, D>& x, unique_ptr<T, D>& y) noexcept;
```

1 *Effects:* Calls `x.swap(y)`.

```
template <class T1, class D1, class T2, class D2>
  bool operator==(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
```

2 *Returns:* `x.get() == y.get()`.

```
template <class T1, class D1, class T2, class D2>
  bool operator!=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
```

3 *Returns:* `x.get() != y.get()`.

```
template <class T1, class D1, class T2, class D2>
  bool operator<(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
```

4 *Requires:* Let `CT` be `common_type<unique_ptr<T1, D1>::pointer, unique_ptr<T2, D2>::pointer>::type`. Then the specialization `less<CT>` shall be a function object type (20.283) that induces a strict weak ordering (25.98) on the pointer values.

5 *Returns:* `less<CT>()(x.get(), y.get())`.

6 *Remarks:* If `unique_ptr<T1, D1>::pointer` is not implicitly convertible to `CT` or `unique_ptr<T2, D2>::pointer` is not implicitly convertible to `CT`, the program is ill-formed.

```
template <class T1, class D1, class T2, class D2>
  bool operator<=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
```

7 *Returns:* `!(y < x)`.

```
template <class T1, class D1, class T2, class D2>
  bool operator>(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
```

8 *Returns:* `y < x`.

```
template <class T1, class D1, class T2, class D2>
  bool operator>=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
```

9 *Returns:* `!(x < y)`.

```
template <class T, class D>
  bool operator==(const unique_ptr<T, D>& x, nullptr_t) noexcept;
template <class T, class D>
  bool operator==(nullptr_t, const unique_ptr<T, D>& x) noexcept;
```

10 *Returns:* `!x`.

```
template <class T, class D>
  bool operator!=(const unique_ptr<T, D>& x, nullptr_t) noexcept;
template <class T, class D>
  bool operator!=(nullptr_t, const unique_ptr<T, D>& x) noexcept;
```

11 *Returns:* `(bool)x`.

```
template <class T, class D>
  bool operator<(const unique_ptr<T, D>& x, nullptr_t);
template <class T, class D>
  bool operator<(nullptr_t, const unique_ptr<T, D>& x);
```

12 *Requires:* The specialization `less<unique_ptr<T, D>::pointer>` shall be a function object type (20.283) that induces a strict weak ordering (25.98) on the pointer values.

13 *Returns:* The first function template returns `less<unique_ptr<T, D>::pointer>()(x.get(), nullptr)`. The second function template returns `less<unique_ptr<T, D>::pointer>()(nullptr, x.get())`.

```
template <class T, class D>
  bool operator>(const unique_ptr<T, D>& x, nullptr_t);
template <class T, class D>
  bool operator>(nullptr_t, const unique_ptr<T, D>& x);
```

14 *Returns:* The first function template returns `nullptr < x`. The second function template returns `x < nullptr`.

```
template <class T, class D>
  bool operator<=(const unique_ptr<T, D>& x, nullptr_t);
template <class T, class D>
  bool operator<=(nullptr_t, const unique_ptr<T, D>& x);
```

15 *Returns:* The first function template returns `!(nullptr < x)`. The second function template returns `!(x < nullptr)`.

```
template <class T, class D>
  bool operator>=(const unique_ptr<T, D>& x, nullptr_t);
template <class T, class D>
  bool operator>=(nullptr_t, const unique_ptr<T, D>& x);
```

16 *Returns:* The first function template returns `!(x < nullptr)`. The second function template returns `!(nullptr < x)`.

20.189.1 Shared-ownership pointers [util.smartptr]

20.189.1.1 Class bad_weak_ptr [util.smartptr.weakptr]

```
namespace std {
  class bad_weak_ptr: public std::exception {
  public:
    bad_weak_ptr() noexcept;
  };
} // namespace std
```

1 An exception of type `bad_weak_ptr` is thrown by the `shared_ptr` constructor taking a `weak_ptr`.

```
bad_weak_ptr() noexcept;
```

2 *Postconditions:* `what()` returns `"bad_weak_ptr"`.

20.189.1.2 Class template shared_ptr [util.smartptr.shared]

1 The `shared_ptr` class template stores a pointer, usually obtained via `new`. `shared_ptr` implements semantics of shared ownership; the last remaining owner of the pointer is responsible for destroying the object, or otherwise releasing the resources associated with the stored pointer. A `shared_ptr` object is *empty* if it does not own a pointer.

```
namespace std {
  template<class T> class shared_ptr {
  public:
    typedef T element_type;
```

```
  // 20.201.0.0.1, constructors:
  constexpr shared_ptr() noexcept;
  template<class Y> explicit shared_ptr(Y* p);
  template<class Y, class D> shared_ptr(Y* p, D d);
  template<class Y, class D, class A> shared_ptr(Y* p, D d, A a);
  template <class D> shared_ptr(nullptr_t p, D d);
  template <class D, class A> shared_ptr(nullptr_t p, D d, A a);
  template<class Y> shared_ptr(const shared_ptr<Y>& r, T* p) noexcept;
  shared_ptr(const shared_ptr& r) noexcept;
  template<class Y> shared_ptr(const shared_ptr<Y>& r) noexcept;
  shared_ptr(shared_ptr&& r) noexcept;
  template<class Y> shared_ptr(shared_ptr<Y>&& r) noexcept;
  template<class Y> explicit shared_ptr(const weak_ptr<Y>& r);
  template<class Y> shared_ptr(auto_ptr<Y>&& r);
  template <class Y, class D> shared_ptr(unique_ptr<Y, D>&& r);
  constexpr shared_ptr(nullptr_t) : shared_ptr() { }

  // 20.229.0.0.1, destructor:
  ~shared_ptr();

  // 20.237.0.0.1, assignment:
  shared_ptr& operator=(const shared_ptr& r) noexcept;
  template<class Y> shared_ptr& operator=(const shared_ptr<Y>& r) noexcept;
  shared_ptr& operator=(shared_ptr&& r) noexcept;
  template<class Y> shared_ptr& operator=(shared_ptr<Y>&& r) noexcept;
  template<class Y> shared_ptr& operator=(auto_ptr<Y>&& r);
  template <class Y, class D> shared_ptr& operator=(unique_ptr<Y, D>&& r);

  // 20.241.0.0.1, modifiers:
  void swap(shared_ptr& r) noexcept;
  void reset() noexcept;
  template<class Y> void reset(Y* p);
  template<class Y, class D> void reset(Y* p, D d);
  template<class Y, class D, class A> void reset(Y* p, D d, A a);

  // 20.241.0.0.2, observers:
  T* get() const noexcept;
  T& operator*() const noexcept;
  T* operator->() const noexcept;
  long use_count() const noexcept;
  bool unique() const noexcept;
  explicit operator bool() const noexcept;
  template<class U> bool owner_before(shared_ptr<U> const& b) const;
  template<class U> bool owner_before(weak_ptr<U> const& b) const;
};

// 20.247.0.0.1, shared_ptr creation
template<class T, class... Args> shared_ptr<T> make_shared(Args&&... args);
template<class T, class A, class... Args>
  shared_ptr<T> allocate_shared(const A& a, Args&&... args);

// 20.262.0.0.1, shared_ptr comparisons:
template<class T, class U>
  bool operator==(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
```

```
  template<class T, class U>
    bool operator!=(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
  template<class T, class U>
    bool operator<(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
  template<class T, class U>
    bool operator>(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
  template<class T, class U>
    bool operator<=(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
  template<class T, class U>
    bool operator>=(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;

  template <class T>
    bool operator==(const shared_ptr<T>& a, nullptr_t) noexcept;
  template <class T>
    bool operator==(nullptr_t, const shared_ptr<T>& b) noexcept;
  template <class T>
    bool operator!=(const shared_ptr<T>& a, nullptr_t) noexcept;
  template <class T>
    bool operator!=(nullptr_t, const shared_ptr<T>& b) noexcept;
  template <class T>
    bool operator<(const shared_ptr<T>& a, nullptr_t) noexcept;
  template <class T>
    bool operator<(nullptr_t, const shared_ptr<T>& b) noexcept;
  template <class T>
    bool operator<=(const shared_ptr<T>& a, nullptr_t) noexcept;
  template <class T>
    bool operator<=(nullptr_t, const shared_ptr<T>& b) noexcept;
  template <class T>
    bool operator>(const shared_ptr<T>& a, nullptr_t) noexcept;
  template <class T>
    bool operator>(nullptr_t, const shared_ptr<T>& b) noexcept;
  template <class T>
    bool operator>=(const shared_ptr<T>& a, nullptr_t) noexcept;
  template <class T>
    bool operator>=(nullptr_t, const shared_ptr<T>& b) noexcept;

  // 20.262.0.0.2, shared_ptr specialized algorithms:
  template<class T> void swap(shared_ptr<T>& a, shared_ptr<T>& b) noexcept;

  // 20.262.0.0.3, shared_ptr casts:
  template<class T, class U>
    shared_ptr<T> static_pointer_cast(const shared_ptr<U>& r) noexcept;
  template<class T, class U>
    shared_ptr<T> dynamic_pointer_cast(const shared_ptr<U>& r) noexcept;
  template<class T, class U>
    shared_ptr<T> const_pointer_cast(const shared_ptr<U>& r) noexcept;

  // 20.263.0.0.1, shared_ptr get_deleter:
  template<class D, class T> D* get_deleter(const shared_ptr<T>& p) noexcept;

  // 20.264.0.0.1, shared_ptr I/O:
  template<class E, class T, class Y>
    basic_ostream<E, T>& operator<< (basic_ostream<E, T>& os, const shared_ptr<Y>& p);
} // namespace std
```

2

Specializations of `shared_ptr` shall be `CopyConstructible`, `CopyAssignable`, and `LessThanComparable`, allowing their use in standard containers. Specializations of `shared_ptr` shall be convertible to `bool`, allowing their use in boolean expressions and declarations in conditions. The template parameter `T` of `shared_ptr` may be an incomplete type.

1 [*Example:*

```
if(shared_ptr<X> px = dynamic_pointer_cast<X>(py)) {
  // do something with px
}
```

— *end example*]

2

For purposes of determining the presence of a data race, member functions shall access and modify only the `shared_ptr` and `weak_ptr` objects themselves and not objects they refer to. Changes in `use_count()` do not reflect modifications that can introduce data races.

20.201.0.0.1 `shared_ptr` constructors [util.smartptr.shared.const]

`constexpr shared_ptr() noexcept;`

1 *Effects:* Constructs an *empty* `shared_ptr` object.

2 *Postconditions:* `use_count() == 0 && get() == nullptr`.

`template<class Y> explicit shared_ptr(Y* p);`

3 *Requires:* `p` shall be convertible to `T*`. `Y` shall be a complete type. The expression `delete p` shall be well formed, shall have well defined behavior, and shall not throw exceptions.

1 *Effects:* Constructs a `shared_ptr` object that *owns* the pointer `p`.

2 *Postconditions:* `use_count() == 1 && get() == p`.

3 *Throws:* `bad_alloc`, or an implementation-defined exception when a resource other than memory could not be obtained.

1 *Exception safety:* If an exception is thrown, `delete p` is called.

```
template<class Y, class D> shared_ptr(Y* p, D d);
template<class Y, class D, class A> shared_ptr(Y* p, D d, A a);
template <class D> shared_ptr(nullptr_t p, D d);
template <class D, class A> shared_ptr(nullptr_t p, D d, A a);
```

1 *Requires:* `p` shall be convertible to `T*`. `D` shall be `CopyConstructible`. The copy constructor and destructor of `D` shall not throw exceptions. The expression `d(p)` shall be well formed, shall have well defined behavior, and shall not throw exceptions. `A` shall be an allocator (17.165.0.1). The copy constructor and destructor of `A` shall not throw exceptions.

1 *Effects:* Constructs a `shared_ptr` object that *owns* the object `p` and the deleter `d`. The second and fourth constructors shall use a copy of `a` to allocate memory for internal use.

1 *Postconditions:* `use_count() == 1 && get() == p`.

2 *Throws:* `bad_alloc`, or an implementation-defined exception when a resource other than memory could not be obtained.

3 *Exception safety:* If an exception is thrown, `d(p)` is called.

`template<class Y> shared_ptr(const shared_ptr<Y>& r, T* p) noexcept;`

1

Effects: Constructs a `shared_ptr` instance that stores `p` and *shares ownership* with `r`.

1 *Postconditions:* `get() == p && use_count() == r.use_count()`

2

[*Note:* To avoid the possibility of a dangling pointer, the user of this constructor must ensure that `p` remains valid at least until the ownership group of `r` is destroyed. — *end note*]

1

[*Note:* This constructor allows creation of an *empty* `shared_ptr` instance with a non-null stored pointer. — *end note*]

```
shared_ptr(const shared_ptr& r) noexcept;
template<class Y> shared_ptr(const shared_ptr<Y>& r) noexcept;
```

1 *Remark:* The second constructor shall not participate in overload resolution unless `Y*` is implicitly convertible to `T*`.

1 *Effects:* If `r` is *empty*, constructs an *empty* `shared_ptr` object; otherwise, constructs a `shared_ptr` object that *shares ownership* with `r`.

1 *Postconditions:* `get() == r.get() && use_count() == r.use_count()`.

```
shared_ptr(shared_ptr&& r) noexcept;
template<class Y> shared_ptr(shared_ptr<Y>&& r) noexcept;
```

1 *Remark:* The second constructor shall not participate in overload resolution unless `Y*` is convertible to `T*`.

2 *Effects:* Move-constructs a `shared_ptr` instance from `r`.

3

Postconditions: `*this` shall contain the old value of `r`. `r` shall be *empty*. `r.get() == nullptr`.

```
template<class Y> explicit shared_ptr(const weak_ptr<Y>& r);
```

1 *Requires:* `Y*` shall be convertible to `T*`.

2 *Effects:* Constructs a `shared_ptr` object that *shares ownership* with `r` and stores a copy of the pointer stored in `r`.

3 *Postconditions:* `use_count() == r.use_count()`.

4 *Throws:* `bad_weak_ptr` when `r.expired()`.

5 *Exception safety:* If an exception is thrown, the constructor has no effect.

```
template<class Y> shared_ptr(auto_ptr<Y>&& r);
```

6 *Requires:* `r.release()` shall be convertible to `T*`. `Y` shall be a complete type. The expression `delete r.release()` shall be well formed, shall have well defined behavior, and shall not throw exceptions.

7 *Effects:* Constructs a `shared_ptr` object that stores and *owns* `r.release()`.

8 *Postconditions:* `use_count() == 1 && r.get() == nullptr`.

9 *Throws:* `bad_alloc`, or an implementation-defined exception when a resource other than memory could not be obtained.

10 *Exception safety:* If an exception is thrown, the constructor has no effect.

```
template <class Y, class D> shared_ptr(unique_ptr<Y, D>&& r);
```

11 *Effects:* Equivalent to `shared_ptr(r.release(), r.get_deleter())` when `D` is not a reference type, otherwise `shared_ptr(r.release(), ref(r.get_deleter()))`.

12 *Exception safety:* If an exception is thrown, the constructor has no effect.

20.229.0.0.1 `shared_ptr` destructor [util.smartptr.shared.dest]

```
~shared_ptr();
```

1 *Effects:*

— If `*this` is *empty* or shares ownership with another `shared_ptr` instance (`use_count() > 1`), there are no side effects.

— Otherwise, if `*this` *owns* an object `p` and a deleter `d`, `d(p)` is called.

— Otherwise, `*this` *owns* a pointer `p`, and `delete p` is called.

1

[*Note:* Since the destruction of `*this` decreases the number of instances that share ownership with `*this` by one, after `*this` has been destroyed all `shared_ptr` instances that shared ownership with `*this` will report a `use_count()` that is one less than its previous value. — *end note*]

20.237.0.0.1 `shared_ptr` assignment [util.smartptr.shared.assign]

```
shared_ptr& operator=(const shared_ptr& r) noexcept;
template<class Y> shared_ptr& operator=(const shared_ptr<Y>& r) noexcept;
template<class Y> shared_ptr& operator=(auto_ptr<Y>&& r);
```

1 *Effects:* Equivalent to `shared_ptr(r).swap(*this)`.

2 *Returns:* `*this`.

3 [*Note:* The use count updates caused by the temporary object construction and destruction are not observable side effects, so the implementation may meet the effects (and the implied guarantees) via different means, without creating a temporary. In particular, in the example:

```
shared_ptr<int> p(new int);
shared_ptr<void> q(p);
p = p;
q = p;
```

both assignments may be no-ops. — *end note*]

```
shared_ptr& operator=(shared_ptr&& r) noexcept;
template<class Y> shared_ptr& operator=(shared_ptr<Y>&& r) noexcept;
```

1 *Effects:* Equivalent to `shared_ptr(std::move(r)).swap(*this)`.

2

Returns: `*this`.

```
template <class Y, class D> shared_ptr& operator=(unique_ptr<Y, D>&& r);
```

1 *Effects:* Equivalent to `shared_ptr(std::move(r)).swap(*this)`.

2 *Returns:* `*this`

20.241.0.0.1 shared_ptr modifiers [util.smartptr.shared.mod]

```
void swap(shared_ptr& r) noexcept;
```

1 *Effects:* Exchanges the contents of `*this` and `r`.

```
void reset() noexcept;
```

2 *Effects:* Equivalent to `shared_ptr().swap(*this)`.

```
template<class Y> void reset(Y* p);
```

3 *Effects:* Equivalent to `shared_ptr(p).swap(*this)`.

```
template<class Y, class D> void reset(Y* p, D d);
```

4 *Effects:* Equivalent to `shared_ptr(p, d).swap(*this)`.

```
template<class Y, class D, class A> void reset(Y* p, D d, A a);
```

5 *Effects:* Equivalent to `shared_ptr(p, d, a).swap(*this)`.

20.241.0.0.2 shared_ptr observers [util.smartptr.shared.obs]

```
T* get() const noexcept;
```

1 *Returns:* the stored pointer.

```
T& operator*() const noexcept;
```

2 *Requires:* `get() != 0`.

3 *Returns:* `*get()`.

4 *Remarks:* When `T` is `void`, it is unspecified whether this member function is declared. If it is declared, it is unspecified what its return type is, except that the declaration (although not necessarily the definition) of the function shall be well formed.

```
T* operator->() const noexcept;
```

5 *Requires:* `get() != 0`.

6 *Returns:* `get()`.

```
long use_count() const noexcept;
```

1 *Returns:* the number of `shared_ptr` objects, `*this` included, that *share ownership* with `*this`, or `0` when `*this` is *empty*.

1 [*Note:* `use_count()` is not necessarily efficient. — *end note*]

```
bool unique() const noexcept;
```

2 *Returns:* `use_count() == 1`.

3 [*Note:* `unique()` may be faster than `use_count()`. If you are using `unique()` to implement copy on write, do not rely on a specific value when `get() == nullptr`. — *end note*]

```
explicit operator bool() const noexcept;
```

1 *Returns:* `get() != 0`.

```
template<class U> bool owner_before(shared_ptr<U> const& b) const;
template<class U> bool owner_before(weak_ptr<U> const& b) const;
```

2 *Returns:* An unspecified value such that

(2.1) — `x.owner_before(y)` defines a strict weak ordering as defined in 25.98;

(2.2) — under the equivalence relation defined by `owner_before`, `!a.owner_before(b) && !b.owner_before(a)`, two `shared_ptr` or `weak_ptr` instances are equivalent if and only if they share ownership or are both empty.

20.247.0.0.1 `shared_ptr` creation [util.smartptr.shared.create]

```
template<class T, class... Args> shared_ptr<T> make_shared(Args&&... args);
template<class T, class A, class... Args>
  shared_ptr<T> allocate_shared(const A& a, Args&&... args);
```

1

Requires: The expression `::new (pv) T(std::forward<Args>(args)...)`, where `pv` has type `void*` and points to storage suitable to hold an object of type `T`, shall be well formed. `A` shall be an *allocator* (17.165.0.1). The copy constructor and destructor of `A` shall not throw exceptions.

1

Effects: Allocates memory suitable for an object of type `T` and constructs an object in that memory via the placement new expression `::new (pv) T(std::forward<Args>(args)...)`. The template `allocate_shared` uses a copy of `a` to allocate memory. If an exception is thrown, the functions have no effect.

1 *Returns:* A `shared_ptr` instance that stores and owns the address of the newly constructed object of type `T`.

2 *Postconditions:* `get() != 0 && use_count() == 1`

3 *Throws:* `bad_alloc`, or an exception thrown from `A::allocate` or from the constructor of `T`.

4

Remarks: Implementations should perform no more than one memory allocation. [*Note:* This provides efficiency equivalent to an intrusive smart pointer. — *end note*]

1

[*Note:* These functions will typically allocate more memory than `sizeof(T)` to allow for internal bookkeeping structures such as the reference counts. — *end note*]

20.262.0.0.1 `shared_ptr` comparison [util.smartptr.shared.cmp]

```
template<class T, class U> bool operator==(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
```

1 *Returns:* `a.get() == b.get()`.

```
template<class T, class U> bool operator<(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
```

2 *Returns:* `less<V>()(a.get(), b.get())`, where `V` is the composite pointer type (Clause 5) of `T*` and `U*`.

3 [*Note:* Defining a comparison operator allows `shared_ptr` objects to be used as keys in associative containers. — *end note*]

```
template <class T>
  bool operator==(const shared_ptr<T>& a, nullptr_t) noexcept;
template <class T>
  bool operator==(nullptr_t, const shared_ptr<T>& a) noexcept;
```

4 *Returns:* `!a`.

```
template <class T>
  bool operator!=(const shared_ptr<T>& a, nullptr_t) noexcept;
template <class T>
  bool operator!=(nullptr_t, const shared_ptr<T>& a) noexcept;
```

5 *Returns:* `(bool)a`.

```
template <class T>
  bool operator<(const shared_ptr<T>& a, nullptr_t) noexcept;
template <class T>
  bool operator<(nullptr_t, const shared_ptr<T>& a) noexcept;
```

6 *Returns:* The first function template returns `less<T*>()(a.get(), nullptr)`. The second function template returns `less<T*>()(nullptr, a.get())`.

```
template <class T>
  bool operator>(const shared_ptr<T>& a, nullptr_t) noexcept;
template <class T>
  bool operator>(nullptr_t, const shared_ptr<T>& a) noexcept;
```

7 *Returns:* The first function template returns `nullptr < a`. The second function template returns `a < nullptr`.

```
template <class T>
  bool operator<=(const shared_ptr<T>& a, nullptr_t) noexcept;
template <class T>
  bool operator<=(nullptr_t, const shared_ptr<T>& a) noexcept;
```

8 *Returns:* The first function template returns `!(nullptr < a)`. The second function template returns `!(a < nullptr)`.

```
template <class T>
  bool operator>=(const shared_ptr<T>& a, nullptr_t) noexcept;
template <class T>
  bool operator>=(nullptr_t, const shared_ptr<T>& a) noexcept;
```

9 *Returns:* The first function template returns `!(a < nullptr)`. The second function template returns `!(nullptr < a)`.

20.262.0.0.2 `shared_ptr` specialized algorithms [util.smartptr.shared.spec]

```
template<class T> void swap(shared_ptr<T>& a, shared_ptr<T>& b) noexcept;
```

1 *Effects:* Equivalent to `a.swap(b)`.

20.262.0.0.3 `shared_ptr` casts [util.smartptr.shared.cast]

```
template<class T, class U> shared_ptr<T> static_pointer_cast(const shared_ptr<U>& r) noexcept;
```

1 *Requires:* The expression `static_cast<T*>(r.get())` shall be well formed.

2 *Returns:* If `r` is *empty*, an *empty* `shared_ptr<T>`; otherwise, a `shared_ptr<T>` object that stores `static_cast<T*>(r.get())` and *shares ownership* with `r`.

3 *Postconditions:* `w.get() == static_cast<T*>(r.get())` and `w.use_count() == r.use_count()`, where `w` is the return value.

4

[*Note:* The seemingly equivalent expression `shared_ptr<T>(static_cast<T*>(r.get()))` will eventually result in undefined behavior, attempting to delete the same object twice. — *end note*]

```
template<class T, class U> shared_ptr<T> dynamic_pointer_cast(const shared_ptr<U>& r) noexcept;
```

1 *Requires:* The expression `dynamic_cast<T*>(r.get())` shall be well formed and shall have well defined behavior.

2 *Returns:*

(2.1) — When `dynamic_cast<T*>(r.get())` returns a nonzero value, a `shared_ptr<T>` object that stores a copy of it and *shares ownership* with `r`;

(2.2) — Otherwise, an *empty* `shared_ptr<T>` object.

3 *Postcondition:* `w.get() == dynamic_cast<T*>(r.get())`, where `w` is the return value.

4 [*Note:* The seemingly equivalent expression `shared_ptr<T>(dynamic_cast<T*>(r.get()))` will eventually result in undefined behavior, attempting to delete the same object twice. — *end note*]

```
template<class T, class U> shared_ptr<T> const_pointer_cast(const shared_ptr<U>& r) noexcept;
```

5 *Requires:* The expression `const_cast<T*>(r.get())` shall be well formed.

6 *Returns:* If `r` is empty, an empty `shared_ptr<T>`; otherwise, a `shared_ptr<T>` object that stores `const_cast<T*>(r.get())` and shares ownership with `r`.

7 *Postconditions:* `w.get() == const_cast<T*>(r.get())` and `w.use_count() == r.use_count()`, where `w` is the return value.

8 [*Note:* The seemingly equivalent expression `shared_ptr<T>(const_cast<T*>(r.get()))` will eventually result in undefined behavior, attempting to delete the same object twice. — *end note*]

20.263.0.0.1 get_deleter [util.smartptr.getdeleter]

```
template<class D, class T> D* get_deleter(const shared_ptr<T>& p) noexcept;
```

1 *Returns:* If `p` *owns* a deleter `d` of type cv-unqualified `D`, returns `&d`; otherwise returns `nullptr`. The returned pointer remains valid as long as there exists a `shared_ptr` instance that owns `d`. [*Note:* It is unspecified whether the pointer remains valid longer than that. This can happen if the implementation doesn't destroy the deleter until all `weak_ptr` instances that share ownership with `p` have been destroyed. — *end note*]

20.264.0.0.1 shared_ptr I/O [util.smartptr.shared.io]

```
template<class E, class T, class Y>
  basic_ostream<E, T>& operator<< (basic_ostream<E, T>& os, shared_ptr<Y> const& p);
```

1 *Effects:* `os << p.get();`.

2 *Returns:* `os`.

20.264.0.1 Class template weak_ptr [util.smartptr.weak]

1 The `weak_ptr` class template stores a weak reference to an object that is already managed by a `shared_ptr`. To access the object, a `weak_ptr` can be converted to a `shared_ptr` using the member function `lock`.

```
namespace std {
  template<class T> class weak_ptr {
  public:
    typedef T element_type;

    // 20.264.0.1.1, constructors
    constexpr weak_ptr() noexcept;
    template<class Y> weak_ptr(shared_ptr<Y> const& r) noexcept;
    weak_ptr(weak_ptr const& r) noexcept;
```

```
    template<class Y> weak_ptr(weak_ptr<Y> const& r) noexcept;
    weak_ptr(weak_ptr&& r) noexcept;
    template<class Y> weak_ptr(weak_ptr<Y>&& r) noexcept;

    // 20.264.0.1.2, destructor
    ~weak_ptr();

    // 20.264.0.1.3, assignment
    weak_ptr& operator=(weak_ptr const& r) noexcept;
    template<class Y> weak_ptr& operator=(weak_ptr<Y> const& r) noexcept;
    template<class Y> weak_ptr& operator=(shared_ptr<Y> const& r) noexcept;
    weak_ptr& operator=(weak_ptr&& r) noexcept;
    template<class Y> weak_ptr& operator=(weak_ptr<Y>&& r) noexcept;

    // 20.264.0.1.4, modifiers
    void swap(weak_ptr& r) noexcept;
    void reset() noexcept;

    // 20.264.0.1.5, observers
    long use_count() const noexcept;
    bool expired() const noexcept;
    shared_ptr<T> lock() const noexcept;
    template<class U> bool owner_before(shared_ptr<U> const& b) const;
    template<class U> bool owner_before(weak_ptr<U> const& b) const;
  };

  // 20.266.0.0.1, specialized algorithms
  template<class T> void swap(weak_ptr<T>& a, weak_ptr<T>& b) noexcept;
} // namespace std
```

2 Specializations of `weak_ptr` shall be `CopyConstructible` and `CopyAssignable`, allowing their use in standard containers. The template parameter `T` of `weak_ptr` may be an incomplete type.

20.264.0.1.1 `weak_ptr` constructors [util.smartptr.weak.const]

```
constexpr weak_ptr() noexcept;
```

1 *Effects:* Constructs an *empty* `weak_ptr` object.

2 *Postconditions:* `use_count() == 0`.

```
weak_ptr(const weak_ptr& r) noexcept;
template<class Y> weak_ptr(const weak_ptr<Y>& r) noexcept;
template<class Y> weak_ptr(const shared_ptr<Y>& r) noexcept;
```

3 *Remark:* The second and third constructors shall not participate in overload resolution unless `Y*` is implicitly convertible to `T*`.

4 *Effects:* If `r` is *empty*, constructs an *empty* `weak_ptr` object; otherwise, constructs a `weak_ptr` object that *shares ownership* with `r` and stores a copy of the pointer stored in `r`.

5 *Postconditions:* `use_count() == r.use_count()`.

```
weak_ptr(weak_ptr&& r) noexcept;
template<class Y> weak_ptr(weak_ptr<Y>&& r) noexcept;
```

6 *Remark:* The second constructor shall not participate in overload resolution unless `Y*` is implicitly convertible to `T*`.

7 *Effects:* Move-constructs a `weak_ptr` instance from `r`.

8 *Postconditions:* `*this` shall contain the old value of `r`. `r` shall be *empty*. `r.use_count() == 0`.

20.264.0.1.2 `weak_ptr` destructor [util.smartptr.weak.dest]

```
~weak_ptr();
```

1 *Effects:* Destroys this `weak_ptr` object but has no effect on the object its stored pointer points to.

20.264.0.1.3 `weak_ptr` assignment [util.smartptr.weak.assign]

```
weak_ptr& operator=(const weak_ptr& r) noexcept;
template<class Y> weak_ptr& operator=(const weak_ptr<Y>& r) noexcept;
template<class Y> weak_ptr& operator=(const shared_ptr<Y>& r) noexcept;
```

1 *Effects:* Equivalent to `weak_ptr(r).swap(*this)`.

2 *Remarks:* The implementation may meet the effects (and the implied guarantees) via different means, without creating a temporary.

3 *Returns:* `*this`.

```
weak_ptr& operator=(weak_ptr&& r) noexcept;
template<class Y> weak_ptr& operator=(weak_ptr<Y>&& r) noexcept;
```

4 *Effects:* Equivalent to `weak_ptr(std::move(r)).swap(*this)`.

5 *Returns:* `*this`.

20.264.0.1.4 `weak_ptr` modifiers [util.smartptr.weak.mod]

```
void swap(weak_ptr& r) noexcept;
```

1 *Effects:* Exchanges the contents of `*this` and `r`.

```
void reset() noexcept;
```

2 *Effects:* Equivalent to `weak_ptr().swap(*this)`.

20.264.0.1.5 `weak_ptr` observers [util.smartptr.weak.obs]

```
long use_count() const noexcept;
```

1 *Returns:* `0` if `*this` is *empty*; otherwise, the number of `shared_ptr` instances that *share ownership* with `*this`.

2 [*Note:* `use_count()` is not necessarily efficient. — *end note*]

```
bool expired() const noexcept;
```

1 *Returns:* `use_count() == 0`.

2 [*Note:* `expired()` may be faster than `use_count()`. — *end note*]

```
shared_ptr<T> lock() const noexcept;
```

3 *Returns:* `expired() ? shared_ptr<T>() : shared_ptr<T>(*this)`, executed atomically.

```
template<class U> bool owner_before(shared_ptr<U> const& b) const;
template<class U> bool owner_before(weak_ptr<U> const& b) const;
```

1 *Returns:* An unspecified value such that

(1.1) — `x.owner_before(y)` defines a strict weak ordering as defined in 25.98;

(1.2) — under the equivalence relation defined by `owner_before`, `!a.owner_before(b) && !b.owner_before(a)`, two `shared_ptr` or `weak_ptr` instances are equivalent if and only if they share ownership or are both empty.

20.266.0.0.1 `weak_ptr` specialized algorithms [util.smartptr.weak.spec]

```
template<class T> void swap(weak_ptr<T>& a, weak_ptr<T>& b) noexcept;
```

1 *Effects:* Equivalent to `a.swap(b)`.

20.266.0.1 Class template `owner_less` [util.smartptr.ownerless]

1 The class template `owner_less` allows ownership-based mixed comparisons of shared and weak pointers.

```
namespace std {
  template<class T> struct owner_less;

  template<class T> struct owner_less<shared_ptr<T> > {
    typedef bool result_type;
    typedef shared_ptr<T> first_argument_type;
    typedef shared_ptr<T> second_argument_type;
    bool operator()(shared_ptr<T> const&, shared_ptr<T> const&) const;
    bool operator()(shared_ptr<T> const&, weak_ptr<T> const&) const;
    bool operator()(weak_ptr<T> const&, shared_ptr<T> const&) const;
  };

  template<class T> struct owner_less<weak_ptr<T> > {
    typedef bool result_type;
    typedef weak_ptr<T> first_argument_type;
    typedef weak_ptr<T> second_argument_type;
    bool operator()(weak_ptr<T> const&, weak_ptr<T> const&) const;
    bool operator()(shared_ptr<T> const&, weak_ptr<T> const&) const;
    bool operator()(weak_ptr<T> const&, shared_ptr<T> const&) const;
  };
}
```

2 `operator()(x,y)` shall return `x.owner_before(y)`. [*Note:* Note that

(2.1) — `operator()` defines a strict weak ordering as defined in 25.98;

(2.2) — under the equivalence relation defined by `operator()`, `!operator()(a, b) && !operator()(b, a)`, two `shared_ptr` or `weak_ptr` instances are equivalent if and only if they share ownership or are both empty.

— *end note*]

20.266.0.2 Class template `enable_shared_from_this` [util.smartptr.enab]

1

A class `T` can inherit from `enable_shared_from_this<T>` to inherit the `shared_from_this` member functions that obtain a *shared_ptr* instance pointing to `*this`.

1 [*Example:*

```
struct X: public enable_shared_from_this<X> {
};

int main() {
  shared_ptr<X> p(new X);
  shared_ptr<X> q = p->shared_from_this();
  assert(p == q);
  assert(!(p < q ) && !(q < p)); // p and q share ownership
}
```

— *end example*]

```
namespace std {
  template<class T> class enable_shared_from_this {
  protected:
    constexpr enable_shared_from_this() noexcept;
    enable_shared_from_this(enable_shared_from_this const&) noexcept;
    enable_shared_from_this& operator=(enable_shared_from_this const&) noexcept;
    ~enable_shared_from_this();
  public:
    shared_ptr<T> shared_from_this();
    shared_ptr<T const> shared_from_this() const;
  };
} // namespace std
```

2 The template parameter `T` of `enable_shared_from_this` may be an incomplete type.

```
constexpr enable_shared_from_this() noexcept;
enable_shared_from_this(const enable_shared_from_this<T>&) noexcept;
```

3 *Effects:* Constructs an `enable_shared_from_this<T>` object.

```
enable_shared_from_this<T>& operator=(const enable_shared_from_this<T>&) noexcept;
```

4 *Returns:* `*this`.

```
~enable_shared_from_this();
```

5 *Effects:* Destroys `*this`.

```
shared_ptr<T>       shared_from_this();
shared_ptr<T const> shared_from_this() const;
```

1 *Requires:* `enable_shared_from_this<T>` shall be an accessible base class of `T`. `*this` shall be a subobject of an object `t` of type `T`. There shall be at least one `shared_ptr` instance `p` that *owns* `&t`.

1 *Returns:* A `shared_ptr<T>` object `r` that *shares ownership with* `p`.

2 *Postconditions:* `r.get() == this`.

3 [*Note:* A possible implementation is shown below:

```
template<class T> class enable_shared_from_this {
private:
  weak_ptr<T> __weak_this;
protected:
  constexpr enable_shared_from_this() : __weak_this() { }
  enable_shared_from_this(enable_shared_from_this const &) { }
  enable_shared_from_this& operator=(enable_shared_from_this const &) { return *this; }
  ~enable_shared_from_this() { }
public:
  shared_ptr<T> shared_from_this() { return shared_ptr<T>(__weak_this); }
  shared_ptr<T const> shared_from_this() const { return shared_ptr<T const>(__weak_this); }
};
```

4

The `shared_ptr` constructors that create unique pointers can detect the presence of an `enable_shared_from_this` base and assign the newly created `shared_ptr` to its `__weak_this` member. — *end note*]

20.279.0.1 shared_ptr atomic access [util.smartptr.shared.atomic]

1

Concurrent access to a `shared_ptr` object from multiple threads does not introduce a data race if the access is done exclusively via the functions in this section and the instance is passed as their first argument.

1 The meaning of the arguments of type `memory_order` is explained in 29.3.

```
template<class T>
  bool atomic_is_lock_free(const shared_ptr<T>* p);
```

2 *Requires:* `p` shall not be null.

3 *Returns:* `true` if atomic access to `*p` is lock-free, `false` otherwise.

4 *Throws:* Nothing.

```
template<class T>
  shared_ptr<T> atomic_load(const shared_ptr<T>* p);
```

5 *Requires:* `p` shall not be null.

6 *Returns:* `atomic_load_explicit(p, memory_order_seq_cst)`.

7 *Throws:* Nothing.

```
template<class T>
  shared_ptr<T> atomic_load_explicit(const shared_ptr<T>* p, memory_order mo);
```

8 *Requires:* `p` shall not be null.

9 *Requires:* `mo` shall not be `memory_order_release` or `memory_order_acq_rel`.

10 *Returns:* `*p`.

11 *Throws:* Nothing.

```
template<class T>
  void atomic_store(shared_ptr<T>* p, shared_ptr<T> r);
```

12 *Requires:* `p` shall not be null.

13 *Effects:* `atomic_store_explicit(p, r, memory_order_seq_cst)`.

14 *Throws:* Nothing.

```
template<class T>
  void atomic_store_explicit(shared_ptr<T>* p, shared_ptr<T> r, memory_order mo);
```

15 *Requires:* `p` shall not be null.

16 *Requires:* `mo` shall not be `memory_order_acquire` or `memory_order_acq_rel`.

17 *Effects:* `p->swap(r)`.

18 *Throws:* Nothing.

```
template<class T>
  shared_ptr<T> atomic_exchange(shared_ptr<T>* p, shared_ptr<T> r);
```

19 *Requires:* `p` shall not be null.

20 *Returns:* `atomic_exchange_explicit(p, r, memory_order_seq_cst)`.

21 *Throws:* Nothing.

```
template<class T>
  shared_ptr<T> atomic_exchange_explicit(shared_ptr<T>* p, shared_ptr<T> r,
                                         memory_order mo);
```

22 *Requires:* `p` shall not be null.

23 *Effects:* `p->swap(r)`.

24 *Returns:* The previous value of `*p`.

25 *Throws:* Nothing.

```
template<class T>
  bool atomic_compare_exchange_weak(
    shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w);
```

26 *Requires:* `p` shall not be null and `v` shall not be null.

27 *Returns:* `atomic_compare_exchange_weak_explicit(p, v, w, memory_order_seq_cst, memory_order_seq_cst)`.

28 *Throws:* Nothing.

```
template<class T>
  bool atomic_compare_exchange_strong(
    shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w);
```

29 *Returns:* `atomic_compare_exchange_strong_explicit(p, v, w, memory_order_seq_cst, memory_order_seq_cst)`.

```
template<class T>
  bool atomic_compare_exchange_weak_explicit(
    shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w,
    memory_order success, memory_order failure);
template<class T>
  bool atomic_compare_exchange_strong_explicit(
    shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w,
    memory_order success, memory_order failure);
```

30 *Requires:* `p` shall not be null and `v` shall not be null.

31 *Requires:* `failure` shall not be `memory_order_release`, `memory_order_acq_rel`, or stronger than `success`.

32 *Effects:* If `*p` is equivalent to `*v`, assigns `w` to `*p` and has synchronization semantics corresponding to the value of `success`, otherwise assigns `*p` to `*v` and has synchronization semantics corresponding to the value of `failure`.

33 *Returns:* `true` if `*p` was equivalent to `*v`, `false` otherwise.

1 *Throws:* Nothing.

2 *Remarks:* two `shared_ptr` objects are equivalent if they store the same pointer value and share ownership.

3 *Remarks:* the weak forms may fail spuriously. See 29.50.

20.282.0.1 Smart pointer hash support [util.smartptr.hash]

```
template <class T, class D> struct hash<unique_ptr<T, D> >;
```

1 The template specialization shall meet the requirements of class template `hash` (20.443.1). For an object `p` of type `UP`, where `UP` is `unique_ptr<T, D>`, `hash<UP>()(p)` shall evaluate to the same value as `hash<typename UP::pointer>()(p.get())`.

2 *Requires:* The specialization `hash<typename UP::pointer>` shall be well-formed and well-defined, and shall meet the requirements of class template `hash` (20.443.1).

```
template <class T> struct hash<shared_ptr<T> >;
```

3 The template specialization shall meet the requirements of class template `hash` (20.443.1). For an object `p` of type `shared_ptr<T>`, `hash<shared_ptr<T> >()(p)` shall evaluate to the same value as `hash<T*>()(p.get())`.

20.283 Function objects [function.objects]

1

A *function object type* is an object type (3.2955) that can be the type of the *postfix-expression* in a function call (5.955.1, 13.230.0.1).[230] A *function object* is an object of a function object type. In the places where one would expect to pass a pointer to a function to an algorithmic template (Clause 25), the interface is specified to accept a function object. This not only makes algorithmic templates work with pointers to functions, but also enables them to work with arbitrary function objects.

1 **Header `<functional>` synopsis**

```
namespace std {
  // D.57.1, base (deprecated):
  template <class Arg, class Result> struct unary_function;
  template <class Arg1, class Arg2, class Result> struct binary_function;

  // 20.339.1, reference_wrapper:
  template <class T> class reference_wrapper;

  template <class T> reference_wrapper<T> ref(T&) noexcept;
  template <class T> reference_wrapper<const T> cref(const T&) noexcept;
  template <class T> void ref(const T&&) = delete;
  template <class T> void cref(const T&&) = delete;

  template <class T> reference_wrapper<T> ref(reference_wrapper<T>) noexcept;
  template <class T> reference_wrapper<const T> cref(reference_wrapper<T>) noexcept;

  // 20.339.2, arithmetic operations:
  template <class T = void> struct plus;
  template <class T = void> struct minus;
  template <class T = void> struct multiplies;
  template <class T = void> struct divides;
  template <class T = void> struct modulus;
  template <class T = void> struct negate;
  template <> struct plus<void>;
  template <> struct minus<void>;
  template <> struct multiplies<void>;
  template <> struct divides<void>;
```

230) Such a type is a function pointer or a class type which has a member `operator()` or a class type which has a conversion to a pointer to function.

```
template <> struct modulus<void>;
template <> struct negate<void>;

// 20.339.3, comparisons:
template <class T = void> struct equal_to;
template <class T = void> struct not_equal_to;
template <class T = void> struct greater;
template <class T = void> struct less;
template <class T = void> struct greater_equal;
template <class T = void> struct less_equal;
template <> struct equal_to<void>;
template <> struct not_equal_to<void>;
template <> struct greater<void>;
template <> struct less<void>;
template <> struct greater_equal<void>;
template <> struct less_equal<void>;

// 20.357.1, logical operations:
template <class T = void> struct logical_and;
template <class T = void> struct logical_or;
template <class T = void> struct logical_not;
template <> struct logical_and<void>;
template <> struct logical_or<void>;
template <> struct logical_not<void>;

// 20.357.2, bitwise operations:
template <class T = void> struct bit_and;
template <class T = void> struct bit_or;
template <class T = void> struct bit_xor;
template <class T = void> struct bit_not;
template <> struct bit_and<void>;
template <> struct bit_or<void>;
template <> struct bit_xor<void>;
template <> struct bit_not<void>;

// 20.357.3, negators:
template <class Predicate> class unary_negate;
template <class Predicate>
  constexpr unary_negate<Predicate> not1(const Predicate&);
template <class Predicate> class binary_negate;
template <class Predicate>
  constexpr binary_negate<Predicate> not2(const Predicate&);

// 20.357.4, bind:
template<class T> struct is_bind_expression;
template<class T> struct is_placeholder;

template<class F, class... BoundArgs>
  unspecified  bind(F&&, BoundArgs&&...);
template<class R, class F, class... BoundArgs>
  unspecified  bind(F&&, BoundArgs&&...);

namespace placeholders {
  // M is the implementation-defined number of placeholders
  extern unspecified  _1;
```

```
  extern unspecified  _2;
              .
              .
              .
  extern unspecified  _M;
}

// D.58, binders (deprecated):
template <class Fn> class binder1st;
template <class Fn, class T>
  binder1st<Fn> bind1st(const Fn&, const T&);
template <class Fn> class binder2nd;
template <class Fn, class T>
  binder2nd<Fn> bind2nd(const Fn&, const T&);

// D.57.2.1, adaptors (deprecated):
template <class Arg, class Result> class pointer_to_unary_function;
template <class Arg, class Result>
  pointer_to_unary_function<Arg,Result> ptr_fun(Result (*)(Arg));
template <class Arg1, class Arg2, class Result>
  class pointer_to_binary_function;
template <class Arg1, class Arg2, class Result>
  pointer_to_binary_function<Arg1,Arg2,Result>
    ptr_fun(Result (*)(Arg1,Arg2));

// D.57.2.2, adaptors (deprecated):
template<class S, class T> class mem_fun_t;
template<class S, class T, class A> class mem_fun1_t;
template<class S, class T>
    mem_fun_t<S,T> mem_fun(S (T::*f)());
template<class S, class T, class A>
    mem_fun1_t<S,T,A> mem_fun(S (T::*f)(A));
template<class S, class T> class mem_fun_ref_t;
template<class S, class T, class A> class mem_fun1_ref_t;
template<class S, class T>
    mem_fun_ref_t<S,T> mem_fun_ref(S (T::*f)());
template<class S, class T, class A>
    mem_fun1_ref_t<S,T,A> mem_fun_ref(S (T::*f)(A));

template <class S, class T> class const_mem_fun_t;
template <class S, class T, class A> class const_mem_fun1_t;
template <class S, class T>
  const_mem_fun_t<S,T> mem_fun(S (T::*f)() const);
template <class S, class T, class A>
  const_mem_fun1_t<S,T,A> mem_fun(S (T::*f)(A) const);
template <class S, class T> class const_mem_fun_ref_t;
template <class S, class T, class A> class const_mem_fun1_ref_t;
template <class S, class T>
  const_mem_fun_ref_t<S,T> mem_fun_ref(S (T::*f)() const);
template <class S, class T, class A>
  const_mem_fun1_ref_t<S,T,A> mem_fun_ref(S (T::*f)(A) const);

// 20.379.1, member function adaptors:
template<class R, class T> unspecified  mem_fn(R T::*);
```

```
  // 20.385.1 polymorphic function wrappers:
  class bad_function_call;

  template<class> class function; // undefined
  template<class R, class... ArgTypes> class function<R(ArgTypes...)>;

  template<class R, class... ArgTypes>
    void swap(function<R(ArgTypes...)>&, function<R(ArgTypes...)>&);

  template<class R, class... ArgTypes>
    bool operator==(const function<R(ArgTypes...)>&, nullptr_t);
  template<class R, class... ArgTypes>
    bool operator==(nullptr_t, const function<R(ArgTypes...)>&);
  template<class R, class... ArgTypes>
    bool operator!=(const function<R(ArgTypes...)>&, nullptr_t);
  template<class R, class... ArgTypes>
    bool operator!=(nullptr_t, const function<R(ArgTypes...)>&);

  // 20.443.1, hash function primary template:
  template <class T> struct hash;

  // Hash function specializations
  template <> struct hash<bool>;
  template <> struct hash<char>;
  template <> struct hash<signed char>;
  template <> struct hash<unsigned char>;
  template <> struct hash<char16_t>;
  template <> struct hash<char32_t>;
  template <> struct hash<wchar_t>;
  template <> struct hash<short>;
  template <> struct hash<unsigned short>;
  template <> struct hash<int>;
  template <> struct hash<unsigned int>;
  template <> struct hash<long>;
  template <> struct hash<long long>;
  template <> struct hash<unsigned long>;
  template <> struct hash<unsigned long long>;

  template <> struct hash<float>;
  template <> struct hash<double>;
  template <> struct hash<long double>;

  template<class T> struct hash<T*>;
}
```

2 [*Example:* If a C++ program wants to have a by-element addition of two vectors `a` and `b` containing `double` and put the result into `a`, it can do:

```
transform(a.begin(), a.end(), b.begin(), a.begin(), plus<double>());
```

— *end example*]

3 [*Example:* To negate every element of `a`:

```
transform(a.begin(), a.end(), a.begin(), negate<double>());
```

— *end example*]

4 [*Note:* To enable adaptors and other components to manipulate function objects that take one or two arguments many of the function objects in this clause correspondingly provide typedefs `argument_type` and `result_type` for function objects that take one argument and `first_argument_type`, `second_argument_type`, and `result_type` for function objects that take two arguments. — *end note*]

20.291.1 Definitions [func.def]

1 The following definitions apply to this Clause:

2 A *call signature* is the name of a return type followed by a parenthesized comma-separated list of zero or more argument types.

1 A *callable type* is a function object type (20.283) or a pointer to member.

1 A *callable object* is an object of a callable type.

2 A *call wrapper type* is a type that holds a callable object and supports a call operation that forwards to that object.

1 A *call wrapper* is an object of a call wrapper type.

1 A *target object* is the callable object held by a call wrapper.

20.295.1 Requirements [func.require]

1 Define *INVOKE*`(f, t1, t2, ..., tN)` as follows:

— `(t1.*f)(t2, ..., tN)` when `f` is a pointer to a member function of a class `T` and `t1` is an object of type `T` or a reference to an object of type `T` or a reference to an object of a type derived from `T`;

— `((*t1).*f)(t2, ..., tN)` when `f` is a pointer to a member function of a class `T` and `t1` is not one of the types described in the previous item;

— `t1.*f` when `N == 1` and `f` is a pointer to member data of a class `T` and `t1` is an object of type `T` or a reference to an object of type `T` or a reference to an object of a type derived from `T`;

— `(*t1).*f` when `N == 1` and `f` is a pointer to member data of a class `T` and `t1` is not one of the types described in the previous item;

— `f(t1, t2, ..., tN)` in all other cases.

1 Define *INVOKE*`(f, t1, t2, ..., tN, R)` as *INVOKE*`(f, t1, t2, ..., tN)` implicitly converted to `R`.

2 If a call wrapper (20.291.1) has a *weak result type* the type of its member type `result_type` is based on the type `T` of the wrapper's target object (20.291.1):

(2.1) — if `T` is a pointer to function type, `result_type` shall be a synonym for the return type of `T`;

(2.2) — if `T` is a pointer to member function, `result_type` shall be a synonym for the return type of `T`;

(2.3) — if `T` is a class type with a member type `result_type`, then `result_type` shall be a synonym for `T::result_type`;

(2.4) — otherwise `result_type` shall not be defined.

3

Every call wrapper (20.291.1) shall be `MoveConstructible`. A *simple call wrapper* is a call wrapper that is `CopyConstructible` and `CopyAssignable` and whose copy constructor, move constructor, and assignment operator do not throw exceptions. A *forwarding call wrapper* is a call wrapper that can be called with an arbitrary argument list and delivers the arguments to the wrapped callable object as references. This forwarding step shall ensure that rvalue arguments are delivered as rvalue-references and lvalue arguments are delivered as lvalue-references. [*Note:* In a typical implementation forwarding call wrappers have an overloaded function call operator of the form

```
template<class... UnBoundArgs>
R operator()(UnBoundArgs&&... unbound_args) cv-qual;
```

— *end note*]

20.339.1 Class template `reference_wrapper` [refwrap]

```
namespace std {
  template <class T> class reference_wrapper {
  public :
    // types
    typedef T type;
    typedef see below result_type;                // not always defined
    typedef see below argument_type;              // not always defined
    typedef see below first_argument_type;        // not always defined
    typedef see below second_argument_type;       // not always defined

    // construct/copy/destroy
    reference_wrapper(T&) noexcept;
    reference_wrapper(T&&) = delete;      // do not bind to temporary objects
    reference_wrapper(const reference_wrapper& x) noexcept;

    // assignment
    reference_wrapper& operator=(const reference_wrapper& x) noexcept;

    // access
    operator T& () const noexcept;
    T& get() const noexcept;

    // invocation
    template <class... ArgTypes>
    result_of_t<T&(ArgTypes&&...)>
    operator() (ArgTypes&&...) const;
  };
}
```

1 `reference_wrapper<T>` is a `CopyConstructible` and `CopyAssignable` wrapper around a reference to an object or function of type `T`.

2 `reference_wrapper<T>` has a weak result type (20.295.1). If `T` is a function type, `result_type` shall be a synonym for the return type of `T`.

3 The template specialization `reference_wrapper<T>` shall define a nested type named `argument_type` as a synonym for `T1` only if the type `T` is any of the following:

(3.1) — a function type or a pointer to function type taking one argument of type `T1`

(3.2) — a pointer to member function `R T0::f` *cv* (where *cv* represents the member function's cv-qualifiers); the type `T1` is *cv* `T0*`

(3.3) — a class type with a member type `argument_type`; the type `T1` is `T::argument_type`.

4 The template instantiation `reference_wrapper<T>` shall define two nested types named `first_argument_type` and `second_argument_type` as synonyms for `T1` and `T2`, respectively, only if the type `T` is any of the following:

(4.1) — a function type or a pointer to function type taking two arguments of types `T1` and `T2`

(4.2) — a pointer to member function `R T0::f(T2)` *cv* (where *cv* represents the member function's cv-qualifiers); the type `T1` is *cv* `T0*`

(4.3) — a class type with member types `first_argument_type` and `second_argument_type`; the type `T1` is `T::first_argument_type` and the type `T2` is `T::second_argument_type`.

20.339.1.1 `reference_wrapper` construct/copy/destroy [refwrap.const]

```
reference_wrapper(T& t) noexcept;
```

1 *Effects:* Constructs a `reference_wrapper` object that stores a reference to `t`.

```
reference_wrapper(const reference_wrapper& x) noexcept;
```

2 *Effects:* Constructs a `reference_wrapper` object that stores a reference to `x.get()`.

20.339.1.2 `reference_wrapper` assignment [refwrap.assign]

```
reference_wrapper& operator=(const reference_wrapper& x) noexcept;
```

1 *Postconditions:* `*this` stores a reference to `x.get()`.

20.339.1.3 `reference_wrapper` access [refwrap.access]

```
operator T& () const noexcept;
```

1 *Returns:* The stored reference.

```
T& get() const noexcept;
```

2 *Returns:* The stored reference.

20.339.1.4 reference_wrapper invocation [refwrap.invoke]

```
template <class... ArgTypes>
  result_of_t<T&(ArgTypes&&... )>
    operator()(ArgTypes&&... args) const;
```

1 *Returns:* *INVOKE*`(get(), std::forward<ArgTypes>(args)...)`. (20.295.1)

2 *Remark:* `operator()` is described for exposition only. Implementations are not required to provide an actual `reference_wrapper::operator()`. Implementations are permitted to support `reference_wrapper` function invocation through multiple overloaded operators or through other means.

20.339.1.5 reference_wrapper helper functions [refwrap.helpers]

```
template <class T> reference_wrapper<T> ref(T& t) noexcept;
```

1 *Returns:* `reference_wrapper<T>(t)`

```
template <class T> reference_wrapper<T> ref(reference_wrapper<T> t) noexcept;
```

2 *Returns:* `ref(t.get())`

```
template <class T> reference_wrapper<const T> cref(const T& t) noexcept;
```

3 *Returns:* `reference_wrapper <const T>(t)`

```
template <class T> reference_wrapper<const T> cref(reference_wrapper<T> t) noexcept;
```

4 *Returns:* `cref(t.get());`

20.339.2 Arithmetic operations [arithmetic.operations]

1 The library provides basic function object classes for all of the arithmetic operators in the language (5.2851, 5.2882).

```
template <class T = void> struct plus {
  constexpr T operator()(const T& x, const T& y) const;
  typedef T first_argument_type;
  typedef T second_argument_type;
  typedef T result_type;
};
```

2 `operator()` returns `x + y`.

```
template <class T = void> struct minus {
  constexpr T operator()(const T& x, const T& y) const;
  typedef T first_argument_type;
  typedef T second_argument_type;
  typedef T result_type;
};
```

3 `operator()` returns `x - y`.

```
template <class T = void> struct multiplies {
  constexpr T operator()(const T& x, const T& y) const;
  typedef T first_argument_type;
  typedef T second_argument_type;
  typedef T result_type;
};
```

4 `operator()` returns `x * y`.

```
template <class T = void> struct divides {
  constexpr T operator()(const T& x, const T& y) const;
  typedef T first_argument_type;
  typedef T second_argument_type;
  typedef T result_type;
};
```

5 `operator()` returns `x / y`.

```
template <class T = void> struct modulus {
  constexpr T operator()(const T& x, const T& y) const;
  typedef T first_argument_type;
  typedef T second_argument_type;
  typedef T result_type;
};
```

6 `operator()` returns `x % y`.

```
template <class T = void> struct negate {
  constexpr T operator()(const T& x) const;
  typedef T argument_type;
  typedef T result_type;
};
```

7 `operator()` returns `-x`.

```
template <> struct plus<void> {
  template <class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) + std::forward<U>(u));

  typedef unspecified  is_transparent;
};
```

8 `operator()` returns `std::forward<T>(t) + std::forward<U>(u)`.

```
template <> struct minus<void> {
  template <class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) - std::forward<U>(u));

  typedef unspecified  is_transparent;
};
```

9 `operator()` returns `std::forward<T>(t) - std::forward<U>(u)`.

```
template <> struct multiplies<void> {
  template <class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) * std::forward<U>(u));

  typedef unspecified  is_transparent;
};
```

10 `operator()` returns `std::forward<T>(t) * std::forward<U>(u)`.

```
template <> struct divides<void> {
  template <class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) / std::forward<U>(u));

  typedef unspecified  is_transparent;
};
```

11 `operator()` returns `std::forward<T>(t) / std::forward<U>(u)`.

```
template <> struct modulus<void> {
  template <class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) % std::forward<U>(u));

  typedef unspecified  is_transparent;
};
```

12 `operator()` returns `std::forward<T>(t) % std::forward<U>(u)`.

```
template <> struct negate<void> {
  template <class T> constexpr auto operator()(T&& t) const
    -> decltype(-std::forward<T>(t));

  typedef unspecified  is_transparent;
};
```

13 `operator()` returns `-std::forward<T>(t)`.

20.339.3 Comparisons [comparisons]

1

The library provides basic function object classes for all of the comparison operators in the language (5.3203, 5.3247).

```
template <class T = void> struct equal_to {
  constexpr bool operator()(const T& x, const T& y) const;
  typedef T first_argument_type;
  typedef T second_argument_type;
  typedef bool result_type;
};
```

1 `operator()` returns `x == y`.

```
template <class T = void> struct not_equal_to {
  constexpr bool operator()(const T& x, const T& y) const;
  typedef T first_argument_type;
  typedef T second_argument_type;
  typedef bool result_type;
};
```

2 `operator()` returns `x != y`.

```
template <class T = void> struct greater {
  constexpr bool operator()(const T& x, const T& y) const;
  typedef T first_argument_type;
  typedef T second_argument_type;
  typedef bool result_type;
};
```

3

`operator()` returns `x > y`.

```
template <class T = void> struct less {
  constexpr bool operator()(const T& x, const T& y) const;
  typedef T first_argument_type;
  typedef T second_argument_type;
  typedef bool result_type;
};
```

1 `operator()` returns `x < y`.

```
template <class T = void> struct greater_equal {
  constexpr bool operator()(const T& x, const T& y) const;
  typedef T first_argument_type;
  typedef T second_argument_type;
  typedef bool result_type;
};
```

2 `operator()` returns `x >= y`.

```
template <class T = void> struct less_equal {
  constexpr bool operator()(const T& x, const T& y) const;
  typedef T first_argument_type;
  typedef T second_argument_type;
  typedef bool result_type;
};
```

3 `operator()` returns `x <= y`.

```
template <> struct equal_to<void> {
  template <class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) == std::forward<U>(u));

  typedef unspecified  is_transparent;
};
```

4 `operator()` returns `std::forward<T>(t) == std::forward<U>(u)`.

```
template <> struct not_equal_to<void> {
  template <class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) != std::forward<U>(u));

  typedef unspecified  is_transparent;
};
```

5 `operator()` returns `std::forward<T>(t) != std::forward<U>(u)`.

```
template <> struct greater<void> {
  template <class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) > std::forward<U>(u));

  typedef unspecified  is_transparent;
};
```

6 `operator()` returns `std::forward<T>(t) > std::forward<U>(u)`.

```
template <> struct less<void> {
  template <class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) < std::forward<U>(u));

  typedef unspecified  is_transparent;
};
```

7 `operator()` returns `std::forward<T>(t) < std::forward<U>(u)`.

```
template <> struct greater_equal<void> {
  template <class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) >= std::forward<U>(u));

  typedef unspecified  is_transparent;
};
```

8 `operator()` returns `std::forward<T>(t) >= std::forward<U>(u)`.

```
template <> struct less_equal<void> {
  template <class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) <= std::forward<U>(u));

  typedef unspecified  is_transparent;
};
```

9 `operator()` returns `std::forward<T>(t) <= std::forward<U>(u)`.

10

For templates `greater`, `less`, `greater_equal`, and `less_equal`, the specializations for any pointer type yield a total order, even if the built-in operators `<`, `>`, `<=`, `>=` do not.

20.357.1 Logical operations [logical.operations]

1 The library provides basic function object classes for all of the logical operators in the language (5.3286, 5.3322, 5.1898.1).

```
template <class T = void> struct logical_and {
  constexpr bool operator()(const T& x, const T& y) const;
  typedef T first_argument_type;
  typedef T second_argument_type;
  typedef bool result_type;
};
```

2 `operator()` returns `x && y`.

```
template <class T = void> struct logical_or {
  constexpr bool operator()(const T& x, const T& y) const;
  typedef T first_argument_type;
  typedef T second_argument_type;
  typedef bool result_type;
};
```

3 `operator()` returns `x || y`.

```
template <class T = void> struct logical_not {
  constexpr bool operator()(const T& x) const;
  typedef T argument_type;
  typedef bool result_type;
};
```

4 `operator()` returns `!x`.

```
template <> struct logical_and<void> {
  template <class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) && std::forward<U>(u));

  typedef unspecified  is_transparent;
};
```

5 `operator()` returns `std::forward<T>(t) && std::forward<U>(u)`.

```
template <> struct logical_or<void> {
  template <class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) || std::forward<U>(u));

  typedef unspecified  is_transparent;
};
```

6 `operator()` returns `std::forward<T>(t) || std::forward<U>(u)`.

```
template <> struct logical_not<void> {
  template <class T> constexpr auto operator()(T&& t) const
    -> decltype(!std::forward<T>(t));

  typedef unspecified  is_transparent;
};
```

7 `operator()` returns `!std::forward<T>(t)`.

20.357.2 Bitwise operations [bitwise.operations]

1 The library provides basic function object classes for all of the bitwise operators in the language (5.3274, 5.3282, 5.3279, 5.1898.1).

```
template <class T = void> struct bit_and {
  constexpr T operator()(const T& x, const T& y) const;
  typedef T first_argument_type;
  typedef T second_argument_type;
  typedef T result_type;
};
```

2 `operator()` returns `x & y`.

```
template <class T = void> struct bit_or {
  constexpr T operator()(const T& x, const T& y) const;
  typedef T first_argument_type;
  typedef T second_argument_type;
  typedef T result_type;
};
```

3 `operator()` returns `x | y`.

```
template <class T = void> struct bit_xor {
  constexpr T operator()(const T& x, const T& y) const;
  typedef T first_argument_type;
  typedef T second_argument_type;
  typedef T result_type;
};
```

4 `operator()` returns `x ^ y`.

```
template <class T = void> struct bit_not {
  constexpr T operator()(const T& x) const;
  typedef T argument_type;
  typedef T result_type;
};
```

5 `operator()` returns `~x`.

```
template <> struct bit_and<void> {
  template <class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) & std::forward<U>(u));

  typedef unspecified  is_transparent;
};
```

6 `operator()` returns `std::forward<T>(t) & std::forward<U>(u)`.

```
template <> struct bit_or<void> {
  template <class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) | std::forward<U>(u));

  typedef unspecified  is_transparent;
};
```

7 `operator()` returns `std::forward<T>(t) | std::forward<U>(u)`.

```
template <> struct bit_xor<void> {
  template <class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) ^ std::forward<U>(u));

  typedef unspecified  is_transparent;
};
```

8 `operator()` returns `std::forward<T>(t) ^ std::forward<U>(u)`.

```
template <> struct bit_not<void> {
  template <class T> constexpr auto operator()(T&& t) const
    -> decltype(~std::forward<T>(t));

  typedef unspecified  is_transparent;
};
```

9 `operator()` returns `~std::forward<T>(t)`.

20.357.3 Negators [negators]

1 Negators `not1` and `not2` take a unary and a binary predicate, respectively, and return their complements (5.1898.1).

```
template <class Predicate>
  class unary_negate {
public:
  constexpr explicit unary_negate(const Predicate& pred);
  constexpr bool operator()(const typename Predicate::argument_type& x) const;
  typedef typename Predicate::argument_type argument_type;
  typedef bool result_type;
};
```

2 `operator()` returns `!pred(x)`.

```
template <class Predicate>
  constexpr unary_negate<Predicate> not1(const Predicate& pred);
```

3 *Returns:* `unary_negate<Predicate>(pred)`.

```
template <class Predicate>
  class binary_negate {
  public:
    constexpr explicit binary_negate(const Predicate& pred);
    constexpr bool operator()(const typename Predicate::first_argument_type& x,
        const typename Predicate::second_argument_type& y) const;
  typedef typename Predicate::first_argument_type first_argument_type;
  typedef typename Predicate::second_argument_type second_argument_type;
  typedef bool result_type;
  };
```

4 `operator()` returns `!pred(x,y)`.

```
template <class Predicate>
  constexpr binary_negate<Predicate> not2(const Predicate& pred);
```

5 *Returns:* `binary_negate<Predicate>(pred)`.

20.357.4 Function object binders [func.bind]

1 This subclause describes a uniform mechanism for binding arguments of callable objects.

20.357.4.1 Class template `is_bind_expression` [func.bind.isbind]

```
namespace std {
  template<class T> struct is_bind_expression; // see below
}
```

1 `is_bind_expression` can be used to detect function objects generated by `bind`. `bind` uses `is_bind_expression` to detect subexpressions.

2 Instantiations of the `is_bind_expression` template shall meet the UnaryTypeTrait requirements (20.450.1). The implementation shall provide a definition that has a BaseCharacteristic of `true_type` if `T` is a type returned from `bind`, otherwise it shall have a BaseCharacteristic of `false_type`. A program may specialize this template for a user-defined type `T` to have a BaseCharacteristic of `true_type` to indicate that `T` should be treated as a subexpression in a `bind` call.

20.357.4.2 Class template `is_placeholder` [func.bind.isplace]

```
namespace std {
  template<class T> struct is_placeholder; // see below
}
```

1 `is_placeholder` can be used to detect the standard placeholders `_1`, `_2`, and so on. `bind` uses `is_placeholder` to detect placeholders.

2

Instantiations of the `is_placeholder` template shall meet the UnaryTypeTrait requirements (20.450.1). The implementation shall provide a definition that has the BaseCharacteristic of `integral_constant<int, J>` if `T` is the type of `std::placeholders::_J`, otherwise it shall have a BaseCharacteristic of `integral_constant<int, 0>`. A program may specialize this template for a user-defined type `T` to have a BaseCharacteristic of `integral_constant<int, N>` with `N > 0` to indicate that `T` should be treated as a placeholder type.

20.358.0.1 Function template `bind` [func.bind.bind]

1

In the text that follows, the following names have the following meanings:

— `FD` is the type `decay_t<F>`,

— `fd` is an lvalue of type `FD` constructed from `std::forward<F>(f)`,

— `Ti` is the ith type in the template parameter pack `BoundArgs`,

— `TiD` is the type `decay_t<Ti>`,

— `ti` is the ith argument in the function parameter pack `bound_args`,

— `tid` is an lvalue of type `TiD` constructed from `std::forward<Ti>(ti)`,

— `Uj` is the jth deduced type of the `UnBoundArgs&&...` parameter of the forwarding call wrapper, and

— `uj` is the jth argument associated with `Uj`.

```
template<class F, class... BoundArgs>
  unspecified  bind(F&& f, BoundArgs&&... bound_args);
```

1

Requires: `is_constructible<FD, F>::value` shall be `true`. For each `Ti` in `BoundArgs`, `is_constructible<TiD, Ti>::value` shall be `true`. *INVOKE* `(fd, w1, w2, ..., wN)` (20.295.1) shall be a valid expression for some values *w1, w2, ..., wN*, where `N == sizeof...(bound_args)`.

1 *Returns:* A forwarding call wrapper `g` with a weak result type (20.295.1). The effect of `g(u1, u2, ..., uM)` shall be *`INVOKE`*`(fd, std::forward<V1>(v1), std::forward<V2>(v2), ..., std::forward<VN>(vN), result_of_t<FD cv & (V1, V2, ..., VN)>)`, where *`cv`* represents the *cv*-qualifiers of `g` and the values and types of the bound arguments `v1, v2, ..., vN` are determined as specified below. The copy constructor and move constructor of the forwarding call wrapper shall throw an exception if and only if the corresponding constructor of `FD` or of any of the types `TiD` throws an exception.

1 *Throws:* Nothing unless the construction of `fd` or of one of the values `tid` throws an exception.

2 *Remarks:* The return type shall satisfy the requirements of `MoveConstructible`. If all of `FD` and `TiD` satisfy the requirements of `CopyConstructible`, then the return type shall satisfy the requirements of `CopyConstructible`. [*Note:* This implies that all of `FD` and `TiD` are `MoveConstructible`. — *end note*]

```
template<class R, class F, class... BoundArgs>
  unspecified  bind(F&& f, BoundArgs&&... bound_args);
```

1 *Requires:* `is_constructible<FD, F>::value` shall be `true`. For each `Ti` in `BoundArgs`, `is_constructible<TiD, Ti>::value` shall be `true`. *`INVOKE`*`(fd, w1, w2, ..., wN)` shall be a valid expression for some values *w1, w2, ..., wN*, where `N == sizeof...(bound_args)`.

2 *Returns:* A forwarding call wrapper `g` with a nested type `result_type` defined as a synonym for `R`. The effect of `g(u1, u2, ..., uM)` shall be *`INVOKE`*`(fd, std::forward<V1>(v1), std::forward<V2>(v2), ..., std::forward<VN>(vN), R)`, where the values and types of the bound arguments `v1, v2, ..., vN` are determined as specified below. The copy constructor and move constructor of the forwarding call wrapper shall throw an exception if and only if the corresponding constructor of `FD` or of any of the types `TiD` throws an exception.

1 *Throws:* Nothing unless the construction of `fd` or of one of the values `tid` throws an exception.

2 *Remarks:* The return type shall satisfy the requirements of `MoveConstructible`. If all of `FD` and `TiD` satisfy the requirements of `CopyConstructible`, then the return type shall satisfy the requirements of `CopyConstructible`. [*Note:* This implies that all of `FD` and `TiD` are `MoveConstructible`. — *end note*]

3 The values of the *bound arguments* `v1, v2, ..., vN` and their corresponding types `V1, V2, ..., VN` depend on the types `TiD` derived from the call to `bind` and the *cv*-qualifiers *cv* of the call wrapper `g` as follows:

— if `TiD` is `reference_wrapper<T>`, the argument is `tid.get()` and its type `Vi` is `T&`;

— if the value of `is_bind_expression<TiD>::value` is `true`, the argument is `tid(std::forward<Uj>(uj)...)` and its type `Vi` is `result_of_t<TiD cv & (Uj&&...)>&&`;

— if the value `j` of `is_placeholder<TiD>::value` is not zero, the argument is `std::forward<Uj>(uj)` and its type `Vi` is `Uj&&`;

— otherwise, the value is `tid` and its type `Vi` is `TiD cv &`.

20.379.0.1 Placeholders [func.bind.place]

```
namespace std {
  namespace placeholders {
    // M is the implementation-defined number of placeholders
```

```
    extern unspecified  _1;
    extern unspecified  _2;
                .
                .
                .
    extern unspecified _M;
  }
}
```

1 All placeholder types shall be `DefaultConstructible` and `CopyConstructible`, and their default constructors and copy/move constructors shall not throw exceptions. It is implementation-defined whether placeholder types are `CopyAssignable`. `CopyAssignable` placeholders' copy assignment operators shall not throw exceptions.

20.379.1 Function template `mem_fn` [func.memfn]

```
template<class R, class T> unspecified  mem_fn(R T::* pm);
```

1

Returns: A simple call wrapper (20.291.1) `fn` such that the expression `fn(t, a2, ..., aN)` is equivalent to *`INVOKE`*`(pm, t, a2, ..., aN)` (20.295.1). `fn` shall have a nested type `result_type` that is a synonym for the return type of `pm` when `pm` is a pointer to member function.

1

The simple call wrapper shall define two nested types named `argument_type` and `result_type` as synonyms for *`cv`* `T*` and `Ret`, respectively, when `pm` is a pointer to member function with cv-qualifier *cv* and taking no arguments, where *Ret* is `pm`'s return type.

1

The simple call wrapper shall define three nested types named `first_argument_type`, `second_argument_type`, and `result_type` as synonyms for *`cv`* `T*`, `T1`, and `Ret`, respectively, when `pm` is a pointer to member function with cv-qualifier *cv* and taking one argument of type `T1`, where *Ret* is `pm`'s return type.

1 *Throws:* Nothing.

20.385.1 Polymorphic function wrappers [func.wrap]

1 This subclause describes a polymorphic wrapper class that encapsulates arbitrary callable objects.

20.385.1.1 Class `bad_function_call` [func.wrap.badcall]

1

An exception of type `bad_function_call` is thrown by `function::operator()` (20.435.0.0.1) when the function wrapper object has no target.

```
namespace std {
  class bad_function_call : public std::exception {
  public:
    // 20.387.0.0.1, constructor:
    bad_function_call() noexcept;
  };
} // namespace std
```

20.387.0.0.1 `bad_function_call` constructor [func.wrap.badcall.const]

```
bad_function_call() noexcept;
```

1 *Effects:* constructs a `bad_function_call` object.

20.387.0.1 Class template function [func.wrap.func]

```
namespace std {
  template<class> class function; // undefined

  template<class R, class... ArgTypes>
  class function<R(ArgTypes...)> {
  public:
    typedef R result_type;
    typedef T1 argument_type;           // only if sizeof...(ArgTypes) == 1 and
                                        // the type in ArgTypes is T1
    typedef T1 first_argument_type;     // only if sizeof...(ArgTypes) == 2 and
                                        // ArgTypes contains T1 and T2
    typedef T2 second_argument_type;    // only if sizeof...(ArgTypes) == 2 and
                                        // ArgTypes contains T1 and T2

    // 20.401.0.0.1, construct/copy/destroy:
    function() noexcept;
    function(nullptr_t) noexcept;
    function(const function&);
    function(function&&);
    template<class F> function(F);
    template<class A> function(allocator_arg_t, const A&) noexcept;
    template<class A> function(allocator_arg_t, const A&,
      nullptr_t) noexcept;
    template<class A> function(allocator_arg_t, const A&,
      const function&);
    template<class A> function(allocator_arg_t, const A&,
      function&&);
    template<class F, class A> function(allocator_arg_t, const A&, F);

    function& operator=(const function&);
    function& operator=(function&&);
    function& operator=(nullptr_t);
    template<class F> function& operator=(F&&);
    template<class F> function& operator=(reference_wrapper<F>) noexcept;

    ~function();

    // 20.434.0.0.1, function modifiers:
    void swap(function&) noexcept;
    template<class F, class A> void assign(F&&, const A&);

    // 20.434.0.0.2, function capacity:
    explicit operator bool() const noexcept;

    // 20.435.0.0.1, function invocation:
    R operator()(ArgTypes...) const;

    // 20.442.0.0.1, function target access:
    const std::type_info& target_type() const noexcept;
    template<class T>       T* target() noexcept;
    template<class T> const T* target() const noexcept;

  };
```

```
  // 20.443.0.0.1, Null pointer comparisons:
  template <class R, class... ArgTypes>
    bool operator==(const function<R(ArgTypes...)>&, nullptr_t) noexcept;

  template <class R, class... ArgTypes>
    bool operator==(nullptr_t, const function<R(ArgTypes...)>&) noexcept;

  template <class R, class... ArgTypes>
    bool operator!=(const function<R(ArgTypes...)>&, nullptr_t) noexcept;

  template <class R, class... ArgTypes>
    bool operator!=(nullptr_t, const function<R(ArgTypes...)>&) noexcept;

  // 20.443.0.0.2, specialized algorithms:
  template <class R, class... ArgTypes>
    void swap(function<R(ArgTypes...)>&, function<R(ArgTypes...)>&);

  template<class R, class... ArgTypes, class Alloc>
    struct uses_allocator<function<R(ArgTypes...)>, Alloc>
      : true_type { };
}
```

1

The `function` class template provides polymorphic wrappers that generalize the notion of a function pointer. Wrappers can store, copy, and call arbitrary callable objects (20.291.1), given a call signature (20.291.1), allowing functions to be first-class objects.

1

A callable object `f` of type `F` is *Callable* for argument types `ArgTypes` and return type `R` if the expression *`INVOKE`*`(f, declval<ArgTypes>()..., R)`, considered as an unevaluated operand (Clause 5), is well formed (20.295.1).

1

The `function` class template is a call wrapper (20.291.1) whose call signature (20.291.1) is `R(ArgTypes...)`.

20.401.0.0.1 `function` construct/copy/destroy [func.wrap.func.con]

1 When any `function` constructor that takes a first argument of type `allocator_arg_t` is invoked, the second argument shall have a type that conforms to the requirements for Allocator (Table 17.165.0.1). A copy of the allocator argument is used to allocate memory, if necessary, for the internal data structures of the constructed `function` object.

```
function() noexcept;
template <class A> function(allocator_arg_t, const A& a) noexcept;
```

2 *Postconditions:* `!*this`.

```
function(nullptr_t) noexcept;
template <class A> function(allocator_arg_t, const A& a, nullptr_t) noexcept;
```

3 *Postconditions:* `!*this`.

```
function(const function& f);
template <class A> function(allocator_arg_t, const A& a, const function& f);
```

4 *Postconditions:* `!*this` if `!f`; otherwise, `*this` targets a copy of `f.target()`.

5

Throws: shall not throw exceptions if `f`'s target is a callable object passed via `reference_wrapper` or a function pointer. Otherwise, may throw `bad_alloc` or any exception thrown by the copy constructor of the stored callable object. [*Note:* Implementations are encouraged to avoid the use of dynamically allocated memory for small callable objects, for example, where `f`'s target is an object holding only a pointer or reference to an object and a member function pointer. — *end note*]

```
function(function&& f);
template <class A> function(allocator_arg_t, const A& a, function&& f);
```

1

Effects: If `!f`, `*this` has no target; otherwise, move-constructs the target of `f` into the target of `*this`, leaving `f` in a valid state with an unspecified value.

```
template<class F> function(F f);
template <class F, class A> function(allocator_arg_t, const A& a, F f);
```

1 *Requires:* `F` shall be `CopyConstructible`.

2

Remarks: These constructors shall not participate in overload resolution unless `f` is Callable (20.387.0.1) for argument types `ArgTypes...` and return type `R`.

1

Postconditions: `!*this` if any of the following hold:

— `f` is a null function pointer value.

— `f` is a null member pointer value.

— `F` is an instance of the `function` class template, and `!f`.

1

Otherwise, `*this` targets a copy of `f` initialized with `std::move(f)`. [*Note:* Implementations are encouraged to avoid the use of dynamically allocated memory for small callable objects, for example, where `f`'s target is an object holding only a pointer or reference to an object and a member function pointer. — *end note*]

1

Throws: shall not throw exceptions when `f` is a function pointer or a `reference_wrapper<T>` for some `T`. Otherwise, may throw `bad_alloc` or any exception thrown by `F`'s copy or move constructor.

```
function& operator=(const function& f);
```

1 *Effects:* `function(f).swap(*this);`

2 *Returns:* `*this`

```
function& operator=(function&& f);
```

3 *Effects:* Replaces the target of `*this` with the target of `f`.

4 *Returns:* `*this`

```
function& operator=(nullptr_t);
```

5 *Effects:* If `*this != nullptr`, destroys the target of `this`.

6 *Postconditions:* `!(*this)`.

7 *Returns:* `*this`

```
template<class F> function& operator=(F&& f);
```

8 *Effects:* `function(std::forward<F>(f)).swap(*this);`

9 *Returns:* `*this`

10 *Remarks:* This assignment operator shall not participate in overload resolution unless `declval<typename decay<F>::type&>()` is Callable (20.387.0.1) for argument types `ArgTypes...` and return type `R`.

```
template<class F> function& operator=(reference_wrapper<F> f) noexcept;
```

11 *Effects:* `function(f).swap(*this);`

12 *Returns:* `*this`

```
~function();
```

13 *Effects:* If `*this != nullptr`, destroys the target of `this`.

20.434.0.0.1 `function` modifiers [func.wrap.func.mod]

```
void swap(function& other) noexcept;
```

1 *Effects:* interchanges the targets of `*this` and `other`.

```
template<class F, class A>
  void assign(F&& f, const A& a);
```

2 *Effects:* `function(allocator_arg, a, std::forward<F>(f)).swap(*this)`

20.434.0.0.2 `function` capacity [func.wrap.func.cap]

```
explicit operator bool() const noexcept;
```

1

Returns: `true` if `*this` has a target, otherwise `false`.

20.435.0.0.1 `function` invocation [func.wrap.func.inv]

```
R operator()(ArgTypes... args) const
```

1

Effects: *`INVOKE`*`(f, std::forward<ArgTypes>(args)..., R)` (20.295.1), where `f` is the target object (20.291.1) of `*this`.

1 *Returns:* Nothing if `R` is `void`, otherwise the return value of *`INVOKE`*`(f, std::forward<ArgTypes>(args)..., R)`.

1 *Throws:* `bad_function_call` if `!*this`; otherwise, any exception thrown by the wrapped callable object.

20.442.0.0.1 function target access [func.wrap.func.targ]

```
const std::type_info& target_type() const noexcept;
```

1 *Returns:* If `*this` has a target of type `T`, `typeid(T)`; otherwise, `typeid(void)`.

```
template<class T>       T* target() noexcept;
template<class T> const T* target() const noexcept;
```

2 *Requires:* `T` shall be a type that is Callable (20.387.0.1) for parameter types `ArgTypes` and return type `R`.

3 *Returns:* If `target_type() == typeid(T)` a pointer to the stored function target; otherwise a null pointer.

20.443.0.0.1 null pointer comparison operators [func.wrap.func.nullptr]

```
template <class R, class... ArgTypes>
  bool operator==(const function<R(ArgTypes...)>& f, nullptr_t) noexcept;
template <class R, class... ArgTypes>
  bool operator==(nullptr_t, const function<R(ArgTypes...)>& f) noexcept;
```

1 *Returns:* `!f`.

```
template <class R, class... ArgTypes>
  bool operator!=(const function<R(ArgTypes...)>& f, nullptr_t) noexcept;
template <class R, class... ArgTypes>
  bool operator!=(nullptr_t, const function<R(ArgTypes...)>& f) noexcept;
```

2 *Returns:* `(bool) f`.

20.443.0.0.2 specialized algorithms [func.wrap.func.alg]

```
template<class R, class... ArgTypes>
  void swap(function<R(ArgTypes...)>& f1, function<R(ArgTypes...)>& f2);
```

1 *Effects:* `f1.swap(f2);`

20.443.1 Class template `hash` [unord.hash]

1

The unordered associative containers defined in 23.932 use specializations of the class template `hash` as the default hash function. For all object types `Key` for which there exists a specialization `hash<Key>`, and for all enumeration types (7.1001) `Key`, the instantiation `hash<Key>` shall:

— satisfy the `Hash` requirements (17.159.0.1), with `Key` as the function call argument type, the `DefaultConstructible` requirements (Table 19), the `CopyAssignable` requirements (Table 23),

— be swappable (17.143.0.1) for lvalues,

— provide two nested types `result_type` and `argument_type` which shall be synonyms for `size_t` and `Key`, respectively,

— satisfy the requirement that if `k1 == k2` is true, `h(k1) == h(k2)` is also true, where `h` is an object of type `hash<Key>` and `k1` and `k2` are objects of type `Key`;

— satisfy the requirement that the expression `h(k)`, where `h` is an object of type `hash<Key>` and `k` is an object of type `Key`, shall not throw an exception unless `hash<Key>` is a user-defined specialization that depends on at least one user-defined type.

```
template <> struct hash<bool>;
template <> struct hash<char>;
template <> struct hash<signed char>;
template <> struct hash<unsigned char>;
template <> struct hash<char16_t>;
template <> struct hash<char32_t>;
template <> struct hash<wchar_t>;
template <> struct hash<short>;
template <> struct hash<unsigned short>;
template <> struct hash<int>;
template <> struct hash<unsigned int>;
template <> struct hash<long>;
template <> struct hash<unsigned long>;
```

```
template <> struct hash<long long>;
template <> struct hash<unsigned long long>;
template <> struct hash<float>;
template <> struct hash<double>;
template <> struct hash<long double>;
template <class T> struct hash<T*>;
```

1 The template specializations shall meet the requirements of class template `hash` (20.443.1).

20.449 Metaprogramming and type traits [meta]

1

This subclause describes components used by C++ programs, particularly in templates, to support the widest possible range of types, optimise template code usage, detect type related user errors, and perform type inference and transformation at compile time. It includes type classification traits, type property inspection traits, and type transformations. The type classification traits describe a complete taxonomy of all possible C++ types, and state where in that taxonomy a given type belongs. The type property inspection traits allow important characteristics of types or of combinations of types to be inspected. The type transformations allow certain properties of types to be manipulated.

20.450.1 Requirements [meta.rqmts]

1

A *UnaryTypeTrait* describes a property of a type. It shall be a class template that takes one template type argument and, optionally, additional arguments that help define the property being described. It shall be `DefaultConstructible`, `CopyConstructible`, and publicly and unambiguously derived, directly or indirectly, from its *BaseCharacteristic*, which is a specialization of the template `integral_constant` (20.457.1), with the arguments to the template `integral_constant` determined by the requirements for the particular property being described. The member names of the BaseCharacteristic shall not be hidden and shall be unambiguously available in the UnaryTypeTrait.

1 A *BinaryTypeTrait* describes a relationship between two types. It shall be a class template that takes two template type arguments and, optionally, additional arguments that help define the relationship being described. It shall be `DefaultConstructible`, `CopyConstructible`, and publicly and unambiguously derived, directly or indirectly, from its *BaseCharacteristic*, which is a specialization of the template `integral_constant` (20.457.1), with the arguments to the template `integral_constant` determined by the requirements for the particular relationship being described. The member names of the BaseCharacteristic shall not be hidden and shall be unambiguously available in the BinaryTypeTrait.

2 A *TransformationTrait* modifies a property of a type. It shall be a class template that takes one template type argument and, optionally, additional arguments that help define the modification. It shall define a publicly accessible nested type named `type`, which shall be a synonym for the modified type.

20.452.1 Header `<type_traits>` synopsis [meta.type.synop]

```
namespace std {
  // 20.457.1, helper class:
  template <class T, T v> struct integral_constant;
  typedef integral_constant<bool, true>  true_type;
  typedef integral_constant<bool, false> false_type;

  // 20.458.0.1, primary type categories:
  template <class T> struct is_void;
  template <class T> struct is_null_pointer;
  template <class T> struct is_integral;
  template <class T> struct is_floating_point;
```

```
template <class T> struct is_array;
template <class T> struct is_pointer;
template <class T> struct is_lvalue_reference;
template <class T> struct is_rvalue_reference;
template <class T> struct is_member_object_pointer;
template <class T> struct is_member_function_pointer;
template <class T> struct is_enum;
template <class T> struct is_union;
template <class T> struct is_class;
template <class T> struct is_function;

// 20.467.0.1, composite type categories:
template <class T> struct is_reference;
template <class T> struct is_arithmetic;
template <class T> struct is_fundamental;
template <class T> struct is_object;
template <class T> struct is_scalar;
template <class T> struct is_compound;
template <class T> struct is_member_pointer;

// 20.467.0.2, type properties:
template <class T> struct is_const;
template <class T> struct is_volatile;
template <class T> struct is_trivial;
template <class T> struct is_trivially_copyable;
template <class T> struct is_standard_layout;
template <class T> struct is_pod;
template <class T> struct is_literal_type;
template <class T> struct is_empty;
template <class T> struct is_polymorphic;
template <class T> struct is_abstract;
template <class T> struct is_final;

template <class T> struct is_signed;
template <class T> struct is_unsigned;

template <class T, class... Args> struct is_constructible;
template <class T> struct is_default_constructible;
template <class T> struct is_copy_constructible;
template <class T> struct is_move_constructible;

template <class T, class U> struct is_assignable;
template <class T> struct is_copy_assignable;
template <class T> struct is_move_assignable;

template <class T> struct is_destructible;

template <class T, class... Args> struct is_trivially_constructible;
template <class T> struct is_trivially_default_constructible;
template <class T> struct is_trivially_copy_constructible;
template <class T> struct is_trivially_move_constructible;

template <class T, class U> struct is_trivially_assignable;
template <class T> struct is_trivially_copy_assignable;
template <class T> struct is_trivially_move_assignable;
```

```
template <class T> struct is_trivially_destructible;

template <class T, class... Args> struct is_nothrow_constructible;
template <class T> struct is_nothrow_default_constructible;
template <class T> struct is_nothrow_copy_constructible;
template <class T> struct is_nothrow_move_constructible;

template <class T, class U> struct is_nothrow_assignable;
template <class T> struct  is_nothrow_copy_assignable;
template <class T> struct is_nothrow_move_assignable;

template <class T> struct is_nothrow_destructible;
template <class T> struct has_virtual_destructor;

// 20.560.1, type property queries:
template <class T> struct alignment_of;
template <class T> struct rank;
template <class T, unsigned I = 0> struct extent;

// 20.561.1, type relations:
template <class T, class U> struct is_same;
template <class Base, class Derived> struct is_base_of;
template <class From, class To> struct is_convertible;

// 20.570.1.1, const-volatile modifications:
template <class T> struct remove_const;
template <class T> struct remove_volatile;
template <class T> struct remove_cv;
template <class T> struct add_const;
template <class T> struct add_volatile;
template <class T> struct add_cv;

template <class T>
  using remove_const_t    = typename remove_const<T>::type;
template <class T>
  using remove_volatile_t = typename remove_volatile<T>::type;
template <class T>
  using remove_cv_t       = typename remove_cv<T>::type;
template <class T>
  using add_const_t       = typename add_const<T>::type;
template <class T>
  using add_volatile_t    = typename add_volatile<T>::type;
template <class T>
  using add_cv_t          = typename add_cv<T>::type;

// 20.570.1.2, reference modifications:
template <class T> struct remove_reference;
template <class T> struct add_lvalue_reference;
template <class T> struct add_rvalue_reference;

template <class T>
  using remove_reference_t     = typename remove_reference<T>::type;
template <class T>
  using add_lvalue_reference_t = typename add_lvalue_reference<T>::type;
template <class T>
```

```
  using add_rvalue_reference_t = typename add_rvalue_reference<T>::type;

// 20.570.1.3, sign modifications:
template <class T> struct make_signed;
template <class T> struct make_unsigned;

template <class T>
  using make_signed_t   = typename make_signed<T>::type;
template <class T>
  using make_unsigned_t = typename make_unsigned<T>::type;

// 20.570.1.4, array modifications:
template <class T> struct remove_extent;
template <class T> struct remove_all_extents;

template <class T>
  using remove_extent_t      = typename remove_extent<T>::type;
template <class T>
  using remove_all_extents_t = typename remove_all_extents<T>::type;

// 20.570.1.5, pointer modifications:
template <class T> struct remove_pointer;
template <class T> struct add_pointer;

template <class T>
  using remove_pointer_t = typename remove_pointer<T>::type;
template <class T>
  using add_pointer_t    = typename add_pointer<T>::type;

// 20.570.1.6, other transformations:
template <std::size_t Len,
          std::size_t Align = default-alignment>   // see 20.570.1.6
  struct aligned_storage;
template <std::size_t Len, class... Types> struct aligned_union;
template <class T> struct decay;
template <bool, class T = void> struct enable_if;
template <bool, class T, class F> struct conditional;
template <class... T> struct common_type;
template <class T> struct underlying_type;
template <class> class result_of;   // not defined
template <class F, class... ArgTypes> class result_of<F(ArgTypes...)>;

template <std::size_t Len,
          std::size_t Align = default-alignment > // see 20.570.1.6
  using aligned_storage_t = typename aligned_storage<Len,Align>::type;
template <std::size_t Len, class... Types>
  using aligned_union_t   = typename aligned_union<Len,Types...>::type;
template <class T>
  using decay_t           = typename decay<T>::type;
template <bool b, class T = void>
  using enable_if_t       = typename enable_if<b,T>::type;
template <bool b, class T, class F>
  using conditional_t     = typename conditional<b,T,F>::type;
template <class... T>
  using common_type_t     = typename common_type<T...>::type;
```

```
  template <class T>
    using underlying_type_t = typename underlying_type<T>::type;
  template <class T>
    using result_of_t       = typename result_of<T>::type;
} // namespace std
```

1

The behavior of a program that adds specializations for any of the class templates defined in this subclause is undefined unless otherwise specified.

20.457.1 Helper classes [meta.help]

```
namespace std {
  template <class T, T v>
  struct integral_constant {
    static constexpr T value = v;
    typedef T value_type;
    typedef integral_constant<T,v> type;
    constexpr operator value_type() const noexcept { return value; }
    constexpr value_type operator()() const noexcept { return value; }
  };
  typedef integral_constant<bool, true> true_type;
  typedef integral_constant<bool, false> false_type;
}
```

1 The class template `integral_constant` and its associated typedefs `true_type` and `false_type` are used as base classes to define the interface for various type traits.

20.457.2 Unary type traits [meta.unary]

1 This sub-clause contains templates that may be used to query the properties of a type at compile time.

2

Each of these templates shall be a UnaryTypeTrait (20.450.1) with a BaseCharacteristic of `true_type` if the corresponding condition is true, otherwise `false_type`.

20.458.0.1 Primary type categories [meta.unary.cat]

1

The primary type categories correspond to the descriptions given in section 3.2955 of the C++ standard.

1 For any given type `T`, the result of applying one of these templates to `T` and to *cv-qualified* `T` shall yield the same result.

2

[*Note:* For any given type `T`, exactly one of the primary type categories has a `value` member that evaluates to `true`. — *end note*]

Table 47 — Primary type category predicates

Template	Condition	Comments
`template <class T>` `struct is_void;`	`T` is `void`	
`template <class T>` `struct is_null_pointer;`	`T` is `std::nullptr_t` (3.3151.1)	

Table 47 — Primary type category predicates (continued)

Template	Condition	Comments
`template <class T>` `struct is_integral;`	`T` is an integral type (3.3151.1)	
`template <class T>` `struct is_floating_point;`	`T` is a floating point type (3.3151.1)	
`template <class T>` `struct is_array;`	`T` is an array type (3.3568.1) of known or unknown extent	Class template `array` (23.542.1) is not an array type.
`template <class T>` `struct is_pointer;`	`T` is a pointer type (3.3568.1)	Includes pointers to functions but not pointers to non-static members.
`template <class T>` `struct is_lvalue_reference;`	`T` is an lvalue reference type (8.97.1)	
`template <class T>` `struct is_rvalue_reference;`	`T` is an rvalue reference type (8.97.1)	
`template <class T>` `struct is_member_object_pointer;`	`T` is a pointer to non-static data member	

`template <class T>`
`struct`
`is_member_function_pointer;`
	`T` is a pointer to non-static member function	

`template <class T>` `struct is_enum;`	`T` is an enumeration type (3.3568.1)	
`template <class T>` `struct is_union;`	`T` is a union type (3.3568.1)	
`template <class T>` `struct is_class;`	`T` is a class type but not a union type (3.3568.1)	
`template <class T>` `struct is_function;`	`T` is a function type (3.3568.1)	

20.467.0.1 Composite type traits [meta.unary.comp]

1 These templates provide convenient compositions of the primary type categories, corresponding to the descriptions given in section 3.2955.

2 For any given type `T`, the result of applying one of these templates to `T`, and to *cv-qualified* `T` shall yield the same result.

Table 48 — Composite type category predicates

Template	Condition	Comments
`template <class T>` `struct is_reference;`	`T` is an lvalue reference or an rvalue reference	
`template <class T>` `struct is_arithmetic;`	`T` is an arithmetic type (3.3151.1)	
`template <class T>` `struct is_fundamental;`	`T` is a fundamental type (3.3151.1)	
`template <class T>` `struct is_object;`	`T` is an object type (3.2955)	
`template <class T>` `struct is_scalar;`	`T` is a scalar type (3.2955)	

Table 48 — Composite type category predicates (continued)

Template	Condition	Comments
`template <class T>` `struct is_compound;`	`T` is a compound type (3.3568.1)	
`template <class T>` `struct is_member_pointer;`	`T` is a pointer to non-static data member or non-static member function	

20.467.0.2 Type properties [meta.unary.prop]

1 These templates provide access to some of the more important properties of types.

2 It is unspecified whether the library defines any full or partial specializations of any of these templates.

3

For all of the class templates `X` declared in this Clause, instantiating that template with a template-argument that is a class template specialization may result in the implicit instantiation of the template argument if and only if the semantics of `X` require that the argument must be a complete type.

Table 49 — Type property predicates

Template	Condition	Preconditions
`template <class T>` `struct is_const;`	`T` is const-qualified (3.3640.1)	
`template <class T>` `struct is_volatile;`	`T` is volatile-qualified (3.3640.1)	
`template <class T>` `struct is_trivial;`	`T` is a trivial type (3.2955)	`remove_all_extents_t<T>` shall be a complete type or (possibly cv-qualified) void.
`template <class T>` `struct is_trivially_copyable;`	`T` is a trivially copyable type (3.2955)	`remove_all_extents_t<T>` shall be a complete type or (possibly cv-qualified) `void`.
`template <class T>` `struct is_standard_layout;`	`T` is a standard-layout type (3.2955)	`remove_all_extents_t<T>` shall be a complete type or (possibly cv-qualified) void.
`template <class T>` `struct is_pod;`	`T` is a POD type (3.2955)	`remove_all_extents_t<T>` shall be a complete type or (possibly cv-qualified) void.
`template <class T>` `struct is_literal_type;`	`T` is a literal type (3.2955)	`remove_all_extents_t<T>` shall be a complete type or (possibly cv-qualified) `void`.

Table 49 — Type property predicates (continued)

Template	Condition	Preconditions
`template <class T>` `struct is_empty;`	`T` is a class type, but not a union type, with no non-static data members other than bit-fields of length 0, no virtual member functions, no virtual base classes, and no base class `B` for which `is_empty<B>::value` is false.	If `T` is a non-union class type, `T` shall be a complete type.
`template <class T>` `struct is_polymorphic;`	`T` is a polymorphic class (10.113)	If `T` is a non-union class type, `T` shall be a complete type.
`template <class T>` `struct is_abstract;`	`T` is an abstract class (10.276)	If `T` is a non-union class type, `T` shall be a complete type.
`template <class T>` `struct is_final;`	`T` is a class type marked with the *class-virt-specifier* `final` (Clause 9). [*Note:* A union is a class type that can be marked with `final`. — *end note*]	If `T` is a class type, `T` shall be a complete type.
`template <class T>` `struct is_signed;`	If `is_arithmetic<T>::value` is `true`, the same result as `integral_constant<bool, T(-1) < T(0)>::value`; otherwise, `false`	
`template <class T>` `struct is_unsigned;`	If `is_arithmetic<T>::value` is `true`, the same result as `integral_constant<bool, T(0) < T(-1)>::value`; otherwise, `false`	
`template <class T, class... Args>` `struct is_constructible;`	*see below*	`T` and all types in the parameter pack `Args` shall be complete types, (possibly cv-qualified) `void`, or arrays of unknown bound.
`template <class T>` `struct is_default_constructible;`	`is_constructible<T>::value` is `true`.	`T` shall be a complete type, (possibly *cv*-qualified) `void`, or an array of unknown bound.

Table 49 — Type property predicates (continued)

Template	Condition	Preconditions
`template <class T>` `struct is_copy_constructible;`	For a referenceable type `T`, the same result as `is_constructible<T, const T&>::value`, otherwise `false`.	`T` shall be a complete type, (possibly *cv*-qualified) `void`, or an array of unknown bound.
`template <class T>` `struct is_move_constructible;`	For a referenceable type `T`, the same result as `is_constructible<T, T&&>::value`, otherwise `false`.	`T` shall be a complete type, (possibly *cv*-qualified) `void`, or an array of unknown bound.
`template <class T, class U>` `struct is_assignable;`	The expression `declval<T>() = declval<U>()` is well-formed when treated as an unevaluated operand (Clause 5). Access checking is performed as if in a context unrelated to `T` and `U`. Only the validity of the immediate context of the assignment expression is considered. [*Note:* The compilation of the expression can result in side effects such as the instantiation of class template specializations and function template specializations, the generation of implicitly-defined functions, and so on. Such side effects are not in the "immediate context" and can result in the program being ill-formed. — *end note*]	`T` and `U` shall be complete types, (possibly cv-qualified) `void`, or arrays of unknown bound.
`template <class T>` `struct is_copy_assignable;`	For a referenceable type `T`, the same result as `is_assignable<T&, const T&>::value`, otherwise `false`.	`T` shall be a complete type, (possibly *cv*-qualified) `void`, or an array of unknown bound.
`template <class T>` `struct is_move_assignable;`	For a referenceable type `T`, the same result as `is_assignable<T&, T&&>::value`, otherwise `false`.	`T` shall be a complete type, (possibly *cv*-qualified) `void`, or an array of unknown bound.

Table 49 — Type property predicates (continued)

Template	Condition	Preconditions
`template <class T>` `struct is_destructible;`	For reference types, `is_destructible<T>::value` is `true`. For incomplete types and function types, `is_destructible<T>::value` is `false`. For object types and given `U` equal to `remove_all_extents_t<T>`, if the expression `std::declval<U&>().~U()` is well-formed when treated as an unevaluated operand (Clause 5), then `is_destructible<T>::value` is `true`, otherwise it is `false`.	`T` shall be a complete type, (possibly *cv*-qualified) `void`, or an array of unknown bound.

`template <class T, class... Args>`
`struct`
`is_trivially_constructible;`
	`is_constructible<T, Args...>::value` is `true` and the variable definition for `is_constructible`, as defined below, is known to call no operation that is not trivial (3.2955, 12).	`T` and all types in the parameter pack `Args` shall be complete types, (possibly cv-qualified) `void`, or arrays of unknown bound.

`template <class T>`
`struct`
`is_trivially_default_constructible;`
	`is_trivially_constructible<T>::value` is `true`.	`T` shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.

`template <class T>`
`struct`
`is_trivially_copy_constructible;`
	For a referenceable type `T`, the same result as `is_trivially_constructible<T, const T&>::value`, otherwise `false`.	`T` shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.

`template <class T>`
`struct`
`is_trivially_move_constructible;`
	For a referenceable type `T`, the same result as `is_trivially_constructible<T, T&&>::value`, otherwise `false`.	`T` shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.

Table 49 — Type property predicates (continued)

Template	Condition	Preconditions
`template <class T, class U> struct is_trivially_assignable;`	`is_assignable<T, U>::value` is `true` and the assignment, as defined by `is_assignable`, is known to call no operation that is not trivial (3.2955, 12).	`T` and `U` shall be complete types, (possibly cv-qualified) `void`, or arrays of unknown bound.
`template <class T> struct is_trivially_copy_assignable;`	For a referenceable type `T`, the same result as `is_trivially_assignable<T&, const T&>::value`, otherwise `false`.	`T` shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.
`template <class T> struct is_trivially_move_assignable;`	For a referenceable type `T`, the same result as `is_trivially_assignable<T&, T&&>::value`, otherwise `false`.	`T` shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.
`template <class T> struct is_trivially_destructible;`	`is_destructible<T>::value` is `true` and the indicated destructor is known to be trivial.	`T` shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.
`template <class T, class... Args> struct is_nothrow_constructible;`	`is_constructible<T, Args...>::value` is `true` and the variable definition for `is_constructible`, as defined below, is known not to throw any exceptions (5.2750.1).	`T` and all types in the parameter pack `Args` shall be complete types, (possibly cv-qualified) `void`, or arrays of unknown bound.
`template <class T> struct is_nothrow_default_constructible;`	`is_nothrow_constructible<T>::value` is `true`.	`T` shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.
`template <class T> struct is_nothrow_copy_constructible;`	For a referenceable type `T`, the same result as `is_nothrow_constructible<T, const T&>::value`, otherwise `false`.	`T` shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.
`template <class T> struct is_nothrow_move_constructible;`	For a referenceable type `T`, the same result as `is_nothrow_constructible<T, T&&>::value`, otherwise `false`.	`T` shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.

Table 49 — Type property predicates (continued)

Template	Condition	Preconditions
`template <class T, class U>` `struct is_nothrow_assignable;`	`is_assignable<T, U>::value` is `true` and the assignment is known not to throw any exceptions (5.2750.1).	`T` and `U` shall be complete types, (possibly cv-qualified) `void`, or arrays of unknown bound.
`template <class T>` `struct is_nothrow_copy_assignable;`	For a referenceable type `T`, the same result as `is_nothrow_assignable<T&, const T&>::value`, otherwise `false`.	`T` shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.
`template <class T>` `struct is_nothrow_move_assignable;`	For a referenceable type `T`, the same result as `is_nothrow_assignable<T&, T&&>::value`, otherwise `false`.	`T` shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.
`template <class T>` `struct is_nothrow_destructible;`	`is_destructible<T>::value` is `true` and the indicated destructor is known not to throw any exceptions (5.2750.1).	`T` shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.
`template <class T>` `struct has_virtual_destructor;`	`T` has a virtual destructor (12.807)	If `T` is a non-union class type, `T` shall be a complete type.

1 [*Example:*

```
is_const<const volatile int>::value      // true
is_const<const int*>::value              // false
is_const<const int&>::value              // false
is_const<int[3]>::value                  // false
is_const<const int[3]>::value            // true
```

— *end example*]

2 [*Example:*

```
remove_const_t<const volatile int>  // volatile int
remove_const_t<const int* const>    // const int*
remove_const_t<const int&>          // const int&
remove_const_t<const int[3]>        // int[3]
```

— *end example*]

3 [*Example:*

```
// Given:
struct P final { };
union U1 { };
union U2 final { };
```

```
// the following assertions hold:
static_assert(!is_final<int>::value, "Error!");
static_assert( is_final<P>::value, "Error!");
static_assert(!is_final<U1>::value, "Error!");
static_assert( is_final<U2>::value, "Error!");
```

— *end example*]

4

Given the following function prototype:

```
template <class T>
  add_rvalue_reference_t<T> create() noexcept;
```

the predicate condition for a template specialization `is_constructible<T, Args...>` shall be satisfied if and only if the following variable definition would be well-formed for some invented variable `t`:

```
T t(create<Args>()...);
```

[*Note:* These tokens are never interpreted as a function declaration. — *end note*] Access checking is performed as if in a context unrelated to `T` and any of the `Args`. Only the validity of the immediate context of the variable initialization is considered. [*Note:* The evaluation of the initialization can result in side effects such as the instantiation of class template specializations and function template specializations, the generation of implicitly-defined functions, and so on. Such side effects are not in the "immediate context" and can result in the program being ill-formed. — *end note*]

20.560.1 Type property queries [meta.unary.prop.query]

1

This sub-clause contains templates that may be used to query properties of types at compile time.

Table 50 — Type property queries

Template	Value
`template <class T>` `struct alignment_of;`	`alignof(T)`. *Requires:* `alignof(T)` shall be a valid expression (5.2747.1)
`template <class T>` `struct rank;`	If `T` names an array type, an integer value representing the number of dimensions of `T`; otherwise, 0.

`template <class T,`
`unsigned I = 0>`
`struct extent;`
	If `T` is not an array type, or if it has rank less than or equal to `I`, or if `I` is 0 and `T` has type "array of unknown bound of `U`", then 0; otherwise, the bound (8.251.1) of the `I`'th dimension of `T`, where indexing of `I` is zero-based

1 Each of these templates shall be a `UnaryTypeTrait` (20.450.1) with a `BaseCharacteristic` of `integral_constant<size_t, Value>`.

2 [*Example:*

```
// the following assertions hold:
assert(rank<int>::value == 0);
assert(rank<int[2]>::value == 1);
assert(rank<int[][4]>::value == 2);
```

— *end example*]

3 [*Example:*

```
// the following assertions hold:
assert(extent<int>::value == 0);
assert(extent<int[2]>::value == 2);
assert(extent<int[2][4]>::value == 2);
assert(extent<int[][4]>::value == 0);
assert((extent<int, 1>::value) == 0);
assert((extent<int[2], 1>::value) == 0);
assert((extent<int[2][4], 1>::value) == 4);
assert((extent<int[][4], 1>::value) == 4);
```

— *end example*]

20.561.1 Relationships between types [meta.rel]

1 This sub-clause contains templates that may be used to query relationships between types at compile time.

2

Each of these templates shall be a BinaryTypeTrait (20.450.1) with a BaseCharacteristic of `true_type` if the corresponding condition is true, otherwise `false_type`.

Table 51 — Type relationship predicates

Template	Condition	Comments
`template <class T, class U>` `struct is_same;`	`T` and `U` name the same type with the same cv-qualifications	
`template <class Base, class Derived>` `struct is_base_of;`	`Base` is a base class of `Derived` (Clause 10) without regard to cv-qualifiers or `Base` and `Derived` are not unions and name the same class type without regard to cv-qualifiers	If `Base` and `Derived` are non-union class types and are different types (ignoring possible cv-qualifiers) then `Derived` shall be a complete type. [*Note:* Base classes that are private, protected, or ambiguous are, nonetheless, base classes. — *end note*]
`template <class From, class To>` `struct is_convertible;`	*see below*	`From` and `To` shall be complete types, arrays of unknown bound, or (possibly cv-qualified) `void` types.

1 [*Example:*

```
struct B {};
struct B1 : B {};
struct B2 : B {};
struct D : private B1, private B2 {};

is_base_of<B, D>::value          // true
is_base_of<const B, D>::value    // true
is_base_of<B, const D>::value    // true
is_base_of<B, const B>::value    // true
is_base_of<D, B>::value          // false
```

```
is_base_of<B&, D&>::value        // false
is_base_of<B[3], D[3]>::value    // false
is_base_of<int, int>::value      // false
```

— *end example*]

2

Given the following function prototype:

```
template <class T>
  add_rvalue_reference_t<T> create() noexcept;
```

the predicate condition for a template specialization `is_convertible<From, To>` shall be satisfied if and only if the return expression in the following code would be well-formed, including any implicit conversions to the return type of the function:

```
To test() {
  return create<From>();
}
```

[*Note:* This requirement gives well defined results for reference types, void types, array types, and function types. — *end note*] Access checking is performed as if in a context unrelated to `To` and `From`. Only the validity of the immediate context of the expression of the *return-statement* (including conversions to the return type) is considered. [*Note:* The evaluation of the conversion can result in side effects such as the instantiation of class template specializations and function template specializations, the generation of implicitly-defined functions, and so on. Such side effects are not in the "immediate context" and can result in the program being ill-formed. — *end note*]

20.570.1 Transformations between types [meta.trans]

1 This sub-clause contains templates that may be used to transform one type to another following some predefined rule.

2 Each of the templates in this subclause shall be a *TransformationTrait* (20.450.1).

20.570.1.1 Const-volatile modifications [meta.trans.cv]

Table 52 — Const-volatile modifications

Template	Comments
`template <class T>` `struct remove_const;`	The member typedef `type` shall name the same type as `T` except that any top-level const-qualifier has been removed. [*Example:* `remove_const_t<const volatile int>` evaluates to `volatile int`, whereas `remove_const_t<const int*>` evaluates to `const int*`. — *end example*]
`template <class T>` `struct remove_volatile;`	The member typedef `type` shall name the same type as `T` except that any top-level volatile-qualifier has been removed. [*Example:* `remove_volatile_t<const volatile int>` evaluates to `const int`, whereas `remove_volatile_t<volatile int*>` evaluates to `volatile int*`. — *end example*]
`template <class T>` `struct remove_cv;`	The member typedef `type` shall be the same as `T` except that any top-level cv-qualifier has been removed. [*Example:* `remove_cv_t<const volatile int>` evaluates to `int`, whereas `remove_cv_t<const volatile int*>` evaluates to `const volatile int*`. — *end example*]

Table 52 — Const-volatile modifications (continued)

Template	Comments
`template <class T>` `struct add_const;`	If `T` is a reference, function, or top-level const-qualified type, then `type` shall name the same type as `T`, otherwise `T const`.
`template <class T>` `struct add_volatile;`	If `T` is a reference, function, or top-level volatile-qualified type, then `type` shall name the same type as `T`, otherwise `T volatile`.
`template <class T>` `struct add_cv;`	The member typedef `type` shall name the same type as `add_const_t<add_volatile_t<T>>`.

20.570.1.2 Reference modifications [meta.trans.ref]

Table 53 — Reference modifications

Template	Comments
`template <class T>` `struct remove_reference;`	If `T` has type "reference to `T1`" then the member typedef `type` shall name `T1`; otherwise, `type` shall name `T`.

`template <class T>`
`struct`
`add_lvalue_reference;`
	If `T` names an object or function type then the member typedef `type` shall name `T&`; otherwise, if `T` names a type "rvalue reference to `T1`" then the member typedef `type` shall name `T1&`; otherwise, `type` shall name `T`.

`template <class T>`
`struct`
`add_rvalue_reference;`
	If `T` names an object or function type then the member typedef `type` shall name `T&&`; otherwise, `type` shall name `T`. [*Note:* This rule reflects the semantics of reference collapsing (8.97.1). For example, when a type `T` names a type `T1&`, the type `add_rvalue_reference_t<T>` is not an rvalue reference. — *end note*]

20.570.1.3 Sign modifications [meta.trans.sign]

Table 54 — Sign modifications

Template	Comments
`template <class T>` `struct make_signed;`	If `T` names a (possibly cv-qualified) signed integer type (3.3151.1) then the member typedef `type` shall name the type `T`; otherwise, if `T` names a (possibly cv-qualified) unsigned integer type then `type` shall name the corresponding signed integer type, with the same cv-qualifiers as `T`; otherwise, `type` shall name the signed integer type with smallest rank (4.807) for which `sizeof(T) == sizeof(type)`, with the same cv-qualifiers as `T`. *Requires:* `T` shall be a (possibly cv-qualified) integral type or enumeration but not a `bool` type.
`template <class T>` `struct make_unsigned;`	If `T` names a (possibly cv-qualified) unsigned integer type (3.3151.1) then the member typedef `type` shall name the type `T`; otherwise, if `T` names a (possibly cv-qualified) signed integer type then `type` shall name the corresponding unsigned integer type, with the same cv-qualifiers as `T`; otherwise, `type` shall name the unsigned integer type with smallest rank (4.807) for which `sizeof(T) == sizeof(type)`, with the same cv-qualifiers as `T`. *Requires:* `T` shall be a (possibly cv-qualified) integral type or enumeration but not a `bool` type.

20.570.1.4 Array modifications [meta.trans.arr]

Table 55 — Array modifications

Template	Comments
`template <class T>` `struct remove_extent;`	If `T` names a type "array of `U`", the member typedef `type` shall be `U`, otherwise `T`. [*Note:* For multidimensional arrays, only the first array dimension is removed. For a type "array of `const U`", the resulting type is `const U`. — *end note*]
`template <class T>` `struct remove_all_extents;`	If `T` is "multi-dimensional array of `U`", the resulting member typedef `type` is `U`, otherwise `T`.

1 [*Example*

```
// the following assertions hold:
assert((is_same<remove_extent_t<int>, int>::value));
assert((is_same<remove_extent_t<int[2]>, int>::value));
assert((is_same<remove_extent_t<int[2][3]>, int[3]>::value));
assert((is_same<remove_extent_t<int[][3]>, int[3]>::value));
```

— *end example*]

2 [*Example*

```
// the following assertions hold:
assert((is_same<remove_all_extents_t<int>, int>::value));
assert((is_same<remove_all_extents_t<int[2]>, int>::value));
assert((is_same<remove_all_extents_t<int[2][3]>, int>::value));
assert((is_same<remove_all_extents_t<int[][3]>, int>::value));
```

— *end example*]

20.570.1.5 Pointer modifications [meta.trans.ptr]

Table 56 — Pointer modifications

Template	Comments
`template <class T>` `struct remove_pointer;`	If `T` has type "(possibly cv-qualified) pointer to `T1`" then the member typedef `type` shall name `T1`; otherwise, it shall name `T`.
`template <class T>` `struct add_pointer;`	The member typedef `type` shall name the same type as `remove_reference_t<T>*`.

20.570.1.6 Other transformations [meta.trans.other]

Table 57 — Other transformations

Template	Condition	Comments
`template <std::size_t Len, std::size_t Align = default-alignment> struct aligned_storage;`	`Len` shall not be zero. `Align` shall be equal to `alignof(T)` for some type `T` or to *default-alignment*.	The value of *default-alignment* shall be the most stringent alignment requirement for any C++ object type whose size is no greater than `Len` (3.2955). The member typedef `type` shall be a POD type suitable for use as uninitialized storage for any object whose size is at most *Len* and whose alignment is a divisor of *Align*.
`template <std::size_t Len, class... Types> struct aligned_union;`	At least one type is provided.	The member typedef `type` shall be a POD type suitable for use as uninitialized storage for any object whose type is listed in `Types`; its size shall be at least `Len`. The static member `alignment_value` shall be an integral constant of type `std::size_t` whose value is the strictest alignment of all types listed in `Types`.
`template <class T> struct decay;`		Let `U` be `remove_reference_t<T>`. If `is_array<U>::value` is `true`, the member typedef `type` shall equal `remove_extent_t<U>*`. If `is_function<U>::value` is `true`, the member typedef `type` shall equal `add_pointer_t<U>`. Otherwise the member typedef `type` equals `remove_cv_t<U>`. [*Note:* This behavior is similar to the lvalue-to-rvalue (4.127), array-to-pointer (4.180), and function-to-pointer (4.220) conversions applied when an lvalue expression is used as an rvalue, but also strips *cv*-qualifiers from class types in order to more closely model by-value argument passing. — *end note*]
`template <bool B, class T = void> struct enable_if;`		If `B` is `true`, the member typedef `type` shall equal `T`; otherwise, there shall be no member `type`.
`template <bool B, class T, class F> struct conditional;`		If `B` is `true`, the member typedef `type` shall equal `T`. If `B` is `false`, the member typedef `type` shall equal `F`.

Table 57 — Other transformations (continued)

Template	Condition	Comments
`template <class... T>` `struct common_type;`		The member typedef `type` shall be defined as set out below. All types in the parameter pack `T` shall be complete or (possibly *cv*) `void`. A program may specialize this trait if at least one template parameter in the specialization is a user-defined type. [*Note:* Such specializations are needed when only explicit conversions are desired among the template arguments. — *end note*]
`template <class T>` `struct underlying_type;`	`T` shall be an enumeration type (7.1001)	The member typedef `type` shall name the underlying type of `T`.

`template <class Fn,`
`class... ArgTypes> struct`
`result_of<Fn(ArgTypes...)>;`
	`Fn` and all types in the parameter pack `ArgTypes` shall be complete types, (possibly cv-qualified) `void`, or arrays of unknown bound.	If the expression `INVOKE(declval<Fn>(), declval<ArgTypes>()...)` is well formed when treated as an unevaluated operand (Clause 5), the member typedef `type` shall name the type `decltype(INVOKE(declval<Fn>(), declval<ArgTypes>()...));` otherwise, there shall be no member `type`. Access checking is performed as if in a context unrelated to `Fn` and `ArgTypes`. Only the validity of the immediate context of the expression is considered. [*Note:* The compilation of the expression can result in side effects such as the instantiation of class template specializations and function template specializations, the generation of implicitly-defined functions, and so on. Such side effects are not in the "immediate context" and can result in the program being ill-formed. — *end note*]

1

[*Note:* A typical implementation would define `aligned_storage` as:

```
template <std::size_t Len, std::size_t Alignment>
struct aligned_storage {
  typedef struct {
    alignas(Alignment) unsigned char __data[Len];
  } type;
};
```

— *end note*]

1

It is implementation-defined whether any extended alignment is supported (3.3924).

1

The nested typedef `common_type::type` shall be defined as follows:

```
template <class ...T> struct common_type;

template <class T>
struct common_type<T> {
  typedef decay_t<T> type;
};

template <class T, class U>
struct common_type<T, U> {
  typedef decay_t<decltype(true ? declval<T>() : declval<U>())> type;
};

template <class T, class U, class... V>
struct common_type<T, U, V...> {
  typedef common_type_t<common_type_t<T, U>, V...> type;
};
```

1 [*Example:* Given these definitions:

```
typedef bool (&PF1)();
typedef short (*PF2)(long);

struct S {
  operator PF2() const;
  double operator()(char, int&);
  void fn(long) const;
  char data;
};

typedef void (S::*PMF)(long) const;
typedef char S::*PMD;
```

the following assertions will hold:

```
static_assert(is_same<result_of_t<S(int)>, short>::value, "Error!");
static_assert(is_same<result_of_t<S&(unsigned char, int&)>, double>::value, "Error!");
static_assert(is_same<result_of_t<PF1()>, bool>::value, "Error!");
static_assert(is_same<result_of_t<PMF(unique_ptr<S>, int)>, void>::value, "Error!");
static_assert(is_same<result_of_t<PMD(S)>, char&&>::value, "Error!");
static_assert(is_same<result_of_t<PMD(const S*)>, const char&>::value, "Error!");
```

— *end example*]

20.575 Compile-time rational arithmetic [ratio]

20.575.1 In general [ratio.general]

1

This subclause describes the ratio library. It provides a class template `ratio` which exactly represents any finite rational number with a numerator and denominator representable by compile-time constants of type `intmax_t`.

1 Throughout this subclause, the names of template parameters are used to express type requirements. If a template parameter is named `R1` or `R2`, and the template argument is not a specialization of the `ratio` template, the program is ill-formed.

20.576.1 Header `<ratio>` synopsis [ratio.syn]

```
namespace std {
  // 20.576.2, class template ratio
  template <intmax_t N, intmax_t D = 1> class ratio;

  // 20.576.3, ratio arithmetic
  template <class R1, class R2> using ratio_add = see below;
  template <class R1, class R2> using ratio_subtract = see below;
  template <class R1, class R2> using ratio_multiply = see below;
  template <class R1, class R2> using ratio_divide = see below;

  // 20.577.1, ratio comparison
  template <class R1, class R2> struct ratio_equal;
  template <class R1, class R2> struct ratio_not_equal;
  template <class R1, class R2> struct ratio_less;
  template <class R1, class R2> struct ratio_less_equal;
  template <class R1, class R2> struct ratio_greater;
  template <class R1, class R2> struct ratio_greater_equal;

  // 20.577.2, convenience SI typedefs
  typedef ratio<1, 1'000'000'000'000'000'000'000'000> yocto;  // see below
  typedef ratio<1,     1'000'000'000'000'000'000'000> zepto;  // see below
  typedef ratio<1,         1'000'000'000'000'000'000> atto;
  typedef ratio<1,             1'000'000'000'000'000> femto;
  typedef ratio<1,                 1'000'000'000'000> pico;
  typedef ratio<1,                     1'000'000'000> nano;
  typedef ratio<1,                         1'000'000> micro;
  typedef ratio<1,                             1'000> milli;
  typedef ratio<1,                               100> centi;
  typedef ratio<1,                                10> deci;
  typedef ratio<                               10, 1> deca;
  typedef ratio<                              100, 1> hecto;
  typedef ratio<                            1'000, 1> kilo;
  typedef ratio<                        1'000'000, 1> mega;
  typedef ratio<                    1'000'000'000, 1> giga;
  typedef ratio<                1'000'000'000'000, 1> tera;
  typedef ratio<            1'000'000'000'000'000, 1> peta;
  typedef ratio<        1'000'000'000'000'000'000, 1> exa;
  typedef ratio<    1'000'000'000'000'000'000'000, 1> zetta;  // see below
  typedef ratio<1'000'000'000'000'000'000'000'000, 1> yotta;  // see below
}
```

20.576.2 Class template `ratio` [ratio.ratio]

```
namespace std {
  template <intmax_t N, intmax_t D = 1>
  class ratio {
```

```
  public:
    static constexpr intmax_t num;
    static constexpr intmax_t den;
    typedef ratio<num, den> type;
  };
}
```

1 If the template argument `D` is zero or the absolute values of either of the template arguments `N` and `D` is not representable by type `intmax_t`, the program is ill-formed. [*Note:* These rules ensure that infinite ratios are avoided and that for any negative input, there exists a representable value of its absolute value which is positive. In a two's complement representation, this excludes the most negative value. — *end note*]

2 The static data members `num` and `den` shall have the following values, where `gcd` represents the greatest common divisor of the absolute values of `N` and `D`:

(2.1) — `num` shall have the value `sign(N) * sign(D) * abs(N) / gcd`.

(2.2) — `den` shall have the value `abs(D) / gcd`.

20.576.3 Arithmetic on `ratios` [ratio.arithmetic]

1 Each of the alias templates `ratio_add`, `ratio_subtract`, `ratio_multiply`, and `ratio_divide` denotes the result of an arithmetic computation on two `ratios` `R1` and `R2`. With `X` and `Y` computed (in the absence of arithmetic overflow) as specified by Table 58, each alias denotes a `ratio<U, V>` such that `U` is the same as `ratio<X, Y>::num` and `V` is the same as `ratio<X, Y>::den`.

2

If it is not possible to represent `U` or `V` with `intmax_t`, the program is ill-formed. Otherwise, an implementation should yield correct values of `U` and `V`. If it is not possible to represent `X` or `Y` with `intmax_t`, the program is ill-formed unless the implementation yields correct values of `U` and `V`.

Table 58 — Expressions used to perform ratio arithmetic

Type	Value of `X`	Value of `Y`
`ratio_add<R1, R2>`	`R1::num * R2::den + R2::num * R1::den`	`R1::den * R2::den`
`ratio_subtract<R1, R2>`	`R1::num * R2::den - R2::num * R1::den`	`R1::den * R2::den`
`ratio_multiply<R1, R2>`	`R1::num * R2::num`	`R1::den * R2::den`
`ratio_divide<R1, R2>`	`R1::num * R2::den`	`R1::den * R2::num`

1 [*Example:*

```
static_assert(ratio_add<ratio<1,3>, ratio<1,6>>::num == 1, "1/3+1/6 == 1/2");
static_assert(ratio_add<ratio<1,3>, ratio<1,6>>::den == 2, "1/3+1/6 == 1/2");
static_assert(ratio_multiply<ratio<1,3>, ratio<3,2>>::num == 1, "1/3*3/2 == 1/2");
static_assert(ratio_multiply<ratio<1,3>, ratio<3,2>>::den == 2, "1/3*3/2 == 1/2");

  // The following cases may cause the program to be ill-formed under some implementations
static_assert(ratio_add<ratio<1,INT_MAX>, ratio<1,INT_MAX>>::num == 2,
  "1/MAX+1/MAX == 2/MAX");
static_assert(ratio_add<ratio<1,INT_MAX>, ratio<1,INT_MAX>>::den == INT_MAX,
  "1/MAX+1/MAX == 2/MAX");
static_assert(ratio_multiply<ratio<1,INT_MAX>, ratio<INT_MAX,2>>::num == 1,
  "1/MAX * MAX/2 == 1/2");
static_assert(ratio_multiply<ratio<1,INT_MAX>, ratio<INT_MAX,2>>::den == 2,
```

```
"1/MAX * MAX/2 == 1/2");
```

— *end example*]

20.577.1 Comparison of ratios [ratio.comparison]

```
template <class R1, class R2> struct ratio_equal
  : integral_constant<bool, see below> { };
```

1 If `R1::num == R2::num` and `R1::den == R2::den`, `ratio_equal<R1, R2>` shall be derived from `integral_constant<bool, true>`; otherwise it shall be derived from `integral_constant<bool, false>`.

```
template <class R1, class R2> struct ratio_not_equal
  : integral_constant<bool, !ratio_equal<R1, R2>::value> { };

template <class R1, class R2> struct ratio_less
  : integral_constant<bool, see below> { };
```

2 If `R1::num * R2::den < R2::num * R1::den`, `ratio_less<R1, R2>` shall be derived from `integral_constant<bool, true>`; otherwise it shall be derived from `integral_constant<bool, false>`. Implementations may use other algorithms to compute this relationship to avoid overflow. If overflow occurs, the program is ill-formed.

```
template <class R1, class R2> struct ratio_less_equal
  : integral_constant<bool, !ratio_less<R2, R1>::value> { };

template <class R1, class R2> struct ratio_greater
  : integral_constant<bool, ratio_less<R2, R1>::value> { };

template <class R1, class R2> struct ratio_greater_equal
  : integral_constant<bool, !ratio_less<R1, R2>::value> { };
```

20.577.2 SI types for ratio [ratio.si]

1 For each of the typedefs `yocto`, `zepto`, `zetta`, and `yotta`, if both of the constants used in its specification are representable by `intmax_t`, the typedef shall be defined; if either of the constants is not representable by `intmax_t`, the typedef shall not be defined.

20.578 Time utilities [time]

20.578.1 In general [time.general]

1 This subclause describes the chrono library (20.578.2) and various C functions (20.605.1) that provide generally useful time utilities.

20.578.2 Header <chrono> synopsis [time.syn]

```
namespace std {
namespace chrono {

// 20.585.2, class template duration
template <class Rep, class Period = ratio<1> > class duration;

// 20.591.1, class template time_point
template <class Clock, class Duration = typename Clock::duration> class time_point;

}  // namespace chrono
```

```
// 20.585.1.3 common_type specializations
template <class Rep1, class Period1, class Rep2, class Period2>
  struct common_type<chrono::duration<Rep1, Period1>, chrono::duration<Rep2, Period2>>;

template <class Clock, class Duration1, class Duration2>
  struct common_type<chrono::time_point<Clock, Duration1>, chrono::time_point<Clock, Duration2>>;

namespace chrono {

// 20.585.1, customization traits
template <class Rep> struct treat_as_floating_point;
template <class Rep> struct duration_values;

// 20.589.0.4, duration arithmetic
template <class Rep1, class Period1, class Rep2, class Period2>
  common_type_t<duration<Rep1, Period1>, duration<Rep2, Period2>>
  constexpr operator+(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
template <class Rep1, class Period1, class Rep2, class Period2>
  common_type_t<duration<Rep1, Period1>, duration<Rep2, Period2>>
  constexpr operator-(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
template <class Rep1, class Period, class Rep2>
  duration<common_type_t<Rep1, Rep2>, Period>
  constexpr operator*(const duration<Rep1, Period>& d, const Rep2& s);
template <class Rep1, class Rep2, class Period>
  duration<common_type_t<Rep1, Rep2>, Period>
  constexpr operator*(const Rep1& s, const duration<Rep2, Period>& d);
template <class Rep1, class Period, class Rep2>
  duration<common_type_t<Rep1, Rep2>, Period>
  constexpr operator/(const duration<Rep1, Period>& d, const Rep2& s);
template <class Rep1, class Period1, class Rep2, class Period2>
  common_type_t<Rep1, Rep2>
  constexpr operator/(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
template <class Rep1, class Period, class Rep2>
  duration<common_type_t<Rep1, Rep2>, Period>
  constexpr operator%(const duration<Rep1, Period>& d, const Rep2& s);
template <class Rep1, class Period1, class Rep2, class Period2>
  common_type_t<duration<Rep1, Period1>, duration<Rep2, Period2>>
  constexpr operator%(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);

// 20.589.0.5, duration comparisons
template <class Rep1, class Period1, class Rep2, class Period2>
  constexpr bool operator==(const duration<Rep1, Period1>& lhs,
                            const duration<Rep2, Period2>& rhs);
template <class Rep1, class Period1, class Rep2, class Period2>
  constexpr bool operator!=(const duration<Rep1, Period1>& lhs,
                            const duration<Rep2, Period2>& rhs);
template <class Rep1, class Period1, class Rep2, class Period2>
  constexpr bool operator< (const duration<Rep1, Period1>& lhs,
                            const duration<Rep2, Period2>& rhs);
template <class Rep1, class Period1, class Rep2, class Period2>
  constexpr bool operator<=(const duration<Rep1, Period1>& lhs,
                            const duration<Rep2, Period2>& rhs);
template <class Rep1, class Period1, class Rep2, class Period2>
  constexpr bool operator> (const duration<Rep1, Period1>& lhs,
                            const duration<Rep2, Period2>& rhs);
```

```
template <class Rep1, class Period1, class Rep2, class Period2>
  constexpr bool operator>=(const duration<Rep1, Period1>& lhs,
                            const duration<Rep2, Period2>& rhs);

// 20.589.0.6, duration_cast
template <class ToDuration, class Rep, class Period>
  constexpr ToDuration duration_cast(const duration<Rep, Period>& d);

// convenience typedefs
typedef duration<signed integer type of at least 64 bits,        nano> nanoseconds;
typedef duration<signed integer type of at least 55 bits,       micro> microseconds;
typedef duration<signed integer type of at least 45 bits,       milli> milliseconds;
typedef duration<signed integer type of at least 35 bits             > seconds;
typedef duration<signed integer type of at least 29 bits, ratio<  60>> minutes;
typedef duration<signed integer type of at least 23 bits, ratio<3600>> hours;

// 20.591.1.5, time_point arithmetic
template <class Clock, class Duration1, class Rep2, class Period2>
  constexpr time_point<Clock, common_type_t<Duration1, duration<Rep2, Period2>>>
  operator+(const time_point<Clock, Duration1>& lhs, const duration<Rep2, Period2>& rhs);
template <class Rep1, class Period1, class Clock, class Duration2>
  constexpr time_point<Clock, common_type_t<duration<Rep1, Period1>, Duration2>>
  operator+(const duration<Rep1, Period1>& lhs, const time_point<Clock, Duration2>& rhs);
template <class Clock, class Duration1, class Rep2, class Period2>
  constexpr time_point<Clock, common_type_t<Duration1, duration<Rep2, Period2>>>
  operator-(const time_point<Clock, Duration1>& lhs, const duration<Rep2, Period2>& rhs);
template <class Clock, class Duration1, class Duration2>
  constexpr common_type_t<Duration1, Duration2>
  operator-(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);

// 20.591.1.6 time_point comparisons
template <class Clock, class Duration1, class Duration2>
   constexpr bool operator==(const time_point<Clock, Duration1>& lhs,
                             const time_point<Clock, Duration2>& rhs);
template <class Clock, class Duration1, class Duration2>
   constexpr bool operator!=(const time_point<Clock, Duration1>& lhs,
                             const time_point<Clock, Duration2>& rhs);
template <class Clock, class Duration1, class Duration2>
   constexpr bool operator< (const time_point<Clock, Duration1>& lhs,
                             const time_point<Clock, Duration2>& rhs);
template <class Clock, class Duration1, class Duration2>
   constexpr bool operator<=(const time_point<Clock, Duration1>& lhs,
                             const time_point<Clock, Duration2>& rhs);
template <class Clock, class Duration1, class Duration2>
   constexpr bool operator> (const time_point<Clock, Duration1>& lhs,
                             const time_point<Clock, Duration2>& rhs);
template <class Clock, class Duration1, class Duration2>
   constexpr bool operator>=(const time_point<Clock, Duration1>& lhs,
                             const time_point<Clock, Duration2>& rhs);

// 20.591.1.7, time_point_cast
template <class ToDuration, class Clock, class Duration>
  constexpr time_point<Clock, ToDuration>
  time_point_cast(const time_point<Clock, Duration>& t);
```

```
// 20.591.2, clocks
class system_clock;
class steady_clock;
class high_resolution_clock;

}  // namespace chrono

inline namespace literals {
inline namespace chrono_literals {

//  20.591.0.1, suffixes for duration literals
constexpr chrono::hours                                  operator "" h(unsigned long long);
constexpr chrono::duration<unspecified , ratio<3600,1>> operator "" h(long double);
constexpr chrono::minutes                                operator "" min(unsigned long long);
constexpr chrono::duration<unspecified , ratio<60,1>>   operator "" min(long double);
constexpr chrono::seconds                                operator "" s(unsigned long long);
constexpr chrono::duration<unspecified >                 operator "" s(long double);
constexpr chrono::milliseconds                           operator "" ms(unsigned long long);
constexpr chrono::duration<unspecified , milli>          operator "" ms(long double);
constexpr chrono::microseconds                           operator "" us(unsigned long long);
constexpr chrono::duration<unspecified , micro>          operator "" us(long double);
constexpr chrono::nanoseconds                            operator "" ns(unsigned long long);
constexpr chrono::duration<unspecified , nano>           operator "" ns(long double);

}  // namespace chrono_literals
}  // namespace literals

namespace chrono {

using namespace literals::chrono_literals;

} // namespace chrono

}  // namespace std
```

20.578.3 Clock requirements [time.clock.req]

1 A clock is a bundle consisting of a `duration`, a `time_point`, and a function `now()` to get the current `time_point`. The origin of the clock's `time_point` is referred to as the clock's *epoch*. A clock shall meet the requirements in Table 59.

2

In Table 59 `C1` and `C2` denote clock types. `t1` and `t2` are values returned by `C1::now()` where the call returning `t1` happens before (1.859) the call returning `t2` and both of these calls occur before `C1::time_point::max()`. [*Note:* this means `C1` did not wrap around between `t1` and `t2`. — *end note*]

Table 59 — Clock requirements

Expression	Return type	Operational semantics
`C1::rep`	An arithmetic type or a class emulating an arithmetic type	The representation type of `C1::duration`.
`C1::period`	a specialization of `ratio`	The tick period of the clock in seconds.

Table 59 — Clock requirements (continued)

Expression	Return type	Operational semantics
`C1::duration`	`chrono::duration<C1::rep, C1::period>`	The `duration` type of the clock.
`C1::time_point`	`chrono::time_point<C1>` or `chrono::time_point<C2, C1::duration>`	The `time_point` type of the clock. `C1` and `C2` shall refer to the same epoch.
`C1::is_steady`	`const bool`	`true` if `t1 <= t2` is always `true` and the time between clock ticks is constant, otherwise `false`.
`C1::now()`	`C1::time_point`	Returns a `time_point` object representing the current point in time.

1

[*Note:* The relative difference in durations between those reported by a given clock and the SI definition is a measure of the quality of implementation. — *end note*]

1

A type `TC` meets the `TrivialClock` requirements if:

— `TC` satisfies the `Clock` requirements (20.578.3),

— the types `TC::rep`, `TC::duration`, and `TC::time_point` satisfy the requirements of `EqualityComparable` (Table 17), `LessThanComparable` (Table 18), `DefaultConstructible` (Table 19), `CopyConstructible` (Table 21), `CopyAssignable` (Table 23), `Destructible` (Table 24), and the requirements of numeric types (26.2). [*Note:* this means, in particular, that operations on these types will not throw exceptions. — *end note*]

— lvalues of the types `TC::rep`, `TC::duration`, and `TC::time_point` are swappable (17.143.0.1),

— the function `TC::now()` does not throw exceptions, and

— the type `TC::time_point::clock` meets the `TrivialClock` requirements, recursively.

20.585.1 Time-related traits [time.traits]

20.585.1.1 `treat_as_floating_point` [time.traits.is_fp]

```
template <class Rep> struct treat_as_floating_point
  : is_floating_point<Rep> { };
```

1 The `duration` template uses the `treat_as_floating_point` trait to help determine if a `duration` object can be converted to another `duration` with a different tick `period`. If `treat_as_floating_point<Rep>::value` is true, then implicit conversions are allowed among `durations`. Otherwise, the implicit convertibility depends on the tick `periods` of the `durations`. [*Note:* The intention of this trait is to indicate whether a given class behaves like a floating-point type, and thus allows division of one value by another with acceptable loss of precision. If `treat_as_floating_point<Rep>::value` is `false`, `Rep` will be treated as if it behaved like an integral type for the purpose of these conversions. — *end note*]

20.585.1.2 `duration_values` [time.traits.duration_values]

```
template <class Rep>
struct duration_values {
public:
  static constexpr Rep zero();
  static constexpr Rep min();
  static constexpr Rep max();
};
```

1 The `duration` template uses the `duration_values` trait to construct special values of the durations representation (`Rep`). This is done because the representation might be a class type with behavior which requires some other implementation to return these special values. In that case, the author of that class type should specialize `duration_values` to return the indicated values.

`static constexpr Rep zero();`

2 *Returns:* `Rep(0)`. [*Note:* `Rep(0)` is specified instead of `Rep()` because `Rep()` may have some other meaning, such as an uninitialized value. — *end note*]

3 *Remark:* The value returned shall be the additive identity.

`static constexpr Rep min();`

4 *Returns:* `numeric_limits<Rep>::lowest()`.

5 *Remark:* The value returned shall compare less than or equal to `zero()`.

`static constexpr Rep max();`

6 *Returns:* `numeric_limits<Rep>::max()`.

7 *Remark:* The value returned shall compare greater than `zero()`.

20.585.1.3 Specializations of `common_type` [time.traits.specializations]

```
template <class Rep1, class Period1, class Rep2, class Period2>
struct common_type<chrono::duration<Rep1, Period1>, chrono::duration<Rep2, Period2>> {
  typedef chrono::duration<common_type_t<Rep1, Rep2>, see below> type;
};
```

1 The `period` of the `duration` indicated by this specialization of `common_type` shall be the greatest common divisor of `Period1` and `Period2`. [*Note:* This can be computed by forming a ratio of the greatest common divisor of `Period1::num` and `Period2::num` and the least common multiple of `Period1::den` and `Period2::den`. — *end note*]

2 [*Note:* The `typedef` name `type` is a synonym for the `duration` with the largest tick `period` possible where both `duration` arguments will convert to it without requiring a division operation. The representation of this type is intended to be able to hold any value resulting from this conversion with no truncation error, although floating-point durations may have round-off errors. — *end note*]

```
template <class Clock, class Duration1, class Duration2>
struct common_type<chrono::time_point<Clock, Duration1>, chrono::time_point<Clock, Duration2>> {
  typedef chrono::time_point<Clock, common_type_t<Duration1, Duration2>> type;
};
```

3 The common type of two `time_point` types is a `time_point` with the same clock as the two types and the common type of their two `durations`.

20.585.2 Class template `duration` [time.duration]

1 A `duration` type measures time between two points in time (`time_points`). A `duration` has a representation which holds a count of ticks and a tick period. The tick period is the amount of time which occurs from one tick to the next, in units of seconds. It is expressed as a rational constant using the template `ratio`.

```
template <class Rep, class Period = ratio<1>>
class duration {
public:
  typedef Rep    rep;
  typedef Period period;
private:
  rep rep_;  // exposition only
public:
  // 20.587.0.1, construct/copy/destroy:
  constexpr duration() = default;
  template <class Rep2>
      constexpr explicit duration(const Rep2& r);
  template <class Rep2, class Period2>
     constexpr duration(const duration<Rep2, Period2>& d);
  ~duration() = default;
  duration(const duration&) = default;
  duration& operator=(const duration&) = default;

  // 20.589.0.1, observer:
  constexpr rep count() const;

  // 20.589.0.2, arithmetic:
  constexpr duration  operator+() const;
  constexpr duration  operator-() const;
  duration& operator++();
  duration  operator++(int);
  duration& operator--();
  duration  operator--(int);

  duration& operator+=(const duration& d);
  duration& operator-=(const duration& d);

  duration& operator*=(const rep& rhs);
  duration& operator/=(const rep& rhs);
  duration& operator%=(const rep& rhs);
  duration& operator%=(const duration& rhs);

  // 20.589.0.3, special values:
  static constexpr duration zero();
  static constexpr duration min();
  static constexpr duration max();
};
```

2 *Requires:* `Rep` shall be an arithmetic type or a class emulating an arithmetic type.

3 *Remarks:* If `duration` is instantiated with a `duration` type for the template argument `Rep`, the program is ill-formed.

4

Remarks: If `Period` is not a specialization of `ratio`, the program is ill-formed.

1 *Remarks:* If `Period::num` is not positive, the program is ill-formed.

2 *Requires:* Members of `duration` shall not throw exceptions other than those thrown by the indicated operations on their representations.

3

Remarks: The defaulted copy constructor of duration shall be a `constexpr` function if and only if the required initialization of the member `rep_` for copy and move, respectively, would satisfy the requirements for a `constexpr` function.

[*Example:*

```
duration<long, ratio<60>> d0;        // holds a count of minutes using a long
duration<long long, milli> d1;       // holds a count of milliseconds using a long long
duration<double, ratio<1, 30>>  d2;  // holds a count with a tick period of 1/30 of a second
                                     // (30 Hz) using a double
```

— *end example*]

20.587.0.1 `duration` constructors [time.duration.cons]

```
template <class Rep2>
  constexpr explicit duration(const Rep2& r);
```

1

Remarks: This constructor shall not participate in overload resolution unless `Rep2` is implicitly convertible to `rep` and

— `treat_as_floating_point<rep>::value` is `true` or

— `treat_as_floating_point<Rep2>::value` is `false`.

[*Example:*

```
duration<int, milli> d(3);          // OK
duration<int, milli> d(3.5);        // error
```

— *end example*]

1 *Effects:* Constructs an object of type `duration`.

2 *Postcondition:* `count() == static_cast<rep>(r)`.

```
template <class Rep2, class Period2>
  constexpr duration(const duration<Rep2, Period2>& d);
```

3 *Remarks:* This constructor shall not participate in overload resolution unless no overflow is induced in the conversion and `treat_as_floating_point<rep>::value` is `true` or both `ratio_divide<Period2, period>::den` is `1` and `treat_as_floating_point<Rep2>::value` is `false`. [*Note:* This requirement prevents implicit truncation error when converting between integral-based `duration` types. Such a construction could easily lead to confusion about the value of the `duration`. — *end note*] [*Example:*

```
duration<int, milli> ms(3);
duration<int, micro> us = ms;       // OK
duration<int, milli> ms2 = us;      // error
```

— *end example*]

4

Effects: Constructs an object of type `duration`, constructing `rep_` from `duration_cast<duration>(d).count()`.

20.589.0.1 `duration` observer [time.duration.observer]

```
constexpr rep count() const;
```

1 *Returns:* `rep_`.

20.589.0.2 duration arithmetic [time.duration.arithmetic]

`constexpr duration operator+() const;`

1 *Returns:* `*this`.

`constexpr duration operator-() const;`

2 *Returns:* `duration(-rep_);`.

`duration& operator++();`

3 *Effects:* `++rep_`.

4 *Returns:* `*this`.

`duration operator++(int);`

5 *Returns:* `duration(rep_++);`.

`duration& operator--();`

6 *Effects:* `--rep_`.

7 *Returns:* `*this`.

`duration operator--(int);`

8 *Returns:* `duration(rep_--);`.

`duration& operator+=(const duration& d);`

9 *Effects:* `rep_ += d.count()`.

10 *Returns:* `*this`.

`duration& operator-=(const duration& d);`

11 *Effects:* `rep_ -= d.count()`.

12 *Returns:* `*this`.

`duration& operator*=(const rep& rhs);`

13 *Effects:* `rep_ *= rhs`.

14 *Returns:* `*this`.

`duration& operator/=(const rep& rhs);`

15 *Effects:* `rep_ /= rhs`.

16 *Returns:* `*this`.

`duration& operator%=(const rep& rhs);`

17 *Effects:* `rep_ %= rhs`.

18 *Returns:* `*this`.

`duration& operator%=(const duration& rhs);`

19 *Effects:* `rep_ %= rhs.count()`.

20 *Returns:* `*this`.

20.589.0.3 duration special values [time.duration.special]

```
static constexpr duration zero();
```

1 *Returns:* `duration(duration_values<rep>::zero())`.

```
static constexpr duration min();
```

2 *Returns:* `duration(duration_values<rep>::min())`.

```
static constexpr duration max();
```

3 *Returns:* `duration(duration_values<rep>::max())`.

20.589.0.4 duration non-member arithmetic [time.duration.nonmember]

1 In the function descriptions that follow, `CD` represents the return type of the function. `CR(A,B)` represents `common_type_t<A, B>`.

```
template <class Rep1, class Period1, class Rep2, class Period2>
  constexpr common_type_t<duration<Rep1, Period1>, duration<Rep2, Period2>>
  operator+(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
```

2 *Returns:* CD(CD(lhs).count() + CD(rhs).count()).

```
template <class Rep1, class Period1, class Rep2, class Period2>
  constexpr common_type_t<duration<Rep1, Period1>, duration<Rep2, Period2>>
  operator-(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
```

3 *Returns:* CD(CD(lhs).count() - CD(rhs).count()).

```
template <class Rep1, class Period, class Rep2>
  constexpr duration<common_type_t<Rep1, Rep2>, Period>
  operator*(const duration<Rep1, Period>& d, const Rep2& s);
```

4 *Remarks:* This operator shall not participate in overload resolution unless `Rep2` is implicitly convertible to `CR(Rep1, Rep2)`.

5 *Returns:* CD(CD(d).count() * s).

```
template <class Rep1, class Rep2, class Period>
  constexpr duration<common_type_t<Rep1, Rep2>, Period>
  operator*(const Rep1& s, const duration<Rep2, Period>& d);
```

6 *Remarks:* This operator shall not participate in overload resolution unless `Rep1` is implicitly convertible to `CR(Rep1, Rep2)`.

7 *Returns:* `d * s`.

```
template <class Rep1, class Period, class Rep2>
  constexpr duration<common_type_t<Rep1, Rep2>, Period>
  operator/(const duration<Rep1, Period>& d, const Rep2& s);
```

8 *Remarks:* This operator shall not participate in overload resolution unless `Rep2` is implicitly convertible to `CR(Rep1, Rep2)` and `Rep2` is not an instantiation of `duration`.

9 *Returns:* CD(CD(d).count() / s).

```
template <class Rep1, class Period1, class Rep2, class Period2>
  constexpr common_type_t<Rep1, Rep2>
  operator/(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
```

10 *Returns:* `CD(lhs).count() / CD(rhs).count()`.

```
template <class Rep1, class Period, class Rep2>
  constexpr duration<common_type_t<Rep1, Rep2>, Period>
  operator%(const duration<Rep1, Period>& d, const Rep2& s);
```

11 *Remarks:* This operator shall not participate in overload resolution unless `Rep2` is implicitly convertible to `CR(Rep1, Rep2)` and `Rep2` is not an instantiation of `duration`.

12 *Returns:* CD(CD(d).count() % s).

```
template <class Rep1, class Period1, class Rep2, class Period2>
  constexpr common_type_t<duration<Rep1, Period1>, duration<Rep2, Period2>>
  operator%(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
```

13 *Returns:* CD(CD(lhs).count() % CD(rhs).count()).

20.589.0.5 `duration` comparisons [time.duration.comparisons]

1 In the function descriptions that follow, `CT` represents `common_type_t<A, B>`, where `A` and `B` are the types of the two arguments to the function.

```
template <class Rep1, class Period1, class Rep2, class Period2>
  constexpr bool operator==(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
```

2 *Returns:* `CT(lhs).count() == CT(rhs).count()`.

```
template <class Rep1, class Period1, class Rep2, class Period2>
  constexpr bool operator!=(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
```

3 *Returns:* `!(lhs == rhs)`.

```
template <class Rep1, class Period1, class Rep2, class Period2>
  constexpr bool operator<(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
```

4 *Returns:* `CT(lhs).count() < CT(rhs).count()`.

```
template <class Rep1, class Period1, class Rep2, class Period2>
  constexpr bool operator<=(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
```

5 *Returns:* `!(rhs < lhs)`.

```
template <class Rep1, class Period1, class Rep2, class Period2>
  constexpr bool operator>(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
```

6 *Returns:* `rhs < lhs`.

```
template <class Rep1, class Period1, class Rep2, class Period2>
  constexpr bool operator>=(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
```

7 *Returns:* `!(lhs < rhs)`.

20.589.0.6 `duration_cast` [time.duration.cast]

```
template <class ToDuration, class Rep, class Period>
  constexpr ToDuration duration_cast(const duration<Rep, Period>& d);
```

1

Remarks: This function shall not participate in overload resolution unless `ToDuration` is an instantiation of `duration`.

1 *Returns:* Let `CF` be `ratio_divide<Period, typename ToDuration::period>`, and `CR` be `common_type< typename ToDuration::rep, Rep, intmax_t>::type`.

(1.1) — If `CF::num == 1` and `CF::den == 1`, returns

```
ToDuration(static_cast<typename ToDuration::rep>(d.count()))
```

(1.2) — otherwise, if `CF::num != 1` and `CF::den == 1`, returns

```
ToDuration(static_cast<typename ToDuration::rep>(
  static_cast<CR>(d.count()) * static_cast<CR>(CF::num)))
```

(1.3) — otherwise, if `CF::num == 1` and `CF::den != 1`, returns

```
ToDuration(static_cast<typename ToDuration::rep>(
  static_cast<CR>(d.count()) / static_cast<CR>(CF::den)))
```

(1.4) — otherwise, returns

```
ToDuration(static_cast<typename ToDuration::rep>(
  static_cast<CR>(d.count()) * static_cast<CR>(CF::num) / static_cast<CR>(CF::den)))
```

Notes: This function does not use any implicit conversions; all conversions are done with `static_cast`. It avoids multiplications and divisions when it is known at compile time that one or more arguments is 1. Intermediate computations are carried out in the widest representation and only converted to the destination representation at the final step.

20.591.0.1 Suffixes for duration literals [time.duration.literals]

1 This section describes literal suffixes for constructing duration literals. The suffixes `h`, `min`, `s`, `ms`, `us`, `ns` denote duration values of the corresponding types `hours`, `minutes`, `seconds`, `milliseconds`, `microseconds`, and `nanoseconds` respectively if they are applied to integral literals.

2 If any of these suffixes are applied to a floating point literal the result is a `chrono::duration` literal with an unspecified floating point representation.

3 If any of these suffixes are applied to an integer literal and the resulting `chrono::duration` value cannot be represented in the result type because of overflow, the program is ill-formed.

4 [*Example:* The following code shows some duration literals.

```
using namespace std::chrono_literals;
auto constexpr aday=24h;
auto constexpr lesson=45min;
auto constexpr halfanhour=0.5h;
```

— *end example*]

```
constexpr chrono::hours                                operator "" h(unsigned long long hours);
constexpr chrono::duration<unspecified , ratio<3600,1>> operator "" h(long double hours);
```

5 *Returns:* A `duration` literal representing `hours` hours.

```
constexpr chrono::minutes                             operator "" min(unsigned long long minutes);
constexpr chrono::duration<unspecified , ratio<60,1>> operator "" min(long double minutes);
```

6 *Returns:* A `duration` literal representing `minutes` minutes.

```
constexpr chrono::seconds                operator "" s(unsigned long long sec);
constexpr chrono::duration<unspecified > operator "" s(long double sec);
```

7 *Returns:* A `duration` literal representing `sec` seconds.

8 [*Note:* The same suffix `s` is used for `basic_string` but there is no conflict, since duration suffixes apply to numbers and string literal suffixes apply to character array literals. — *end note*]

```
constexpr chrono::milliseconds                  operator "" ms(unsigned long long msec);
constexpr chrono::duration<unspecified , milli> operator "" ms(long double msec);
```

9 *Returns:* A `duration` literal representing `msec` milliseconds.

```
constexpr chrono::microseconds                  operator "" us(unsigned long long usec);
constexpr chrono::duration<unspecified , micro> operator "" us(long double usec);
```

10 *Returns:* A `duration` literal representing `usec` microseconds.

```
constexpr chrono::nanoseconds                  operator "" ns(unsigned long long nsec);
constexpr chrono::duration<unspecified , nano> operator "" ns(long double nsec);
```

11 *Returns:* A `duration` literal representing `nsec` nanoseconds.

20.591.1 Class template `time_point` [time.point]

```
template <class Clock, class Duration = typename Clock::duration>
class time_point {
public:
  typedef Clock                     clock;
  typedef Duration                  duration;
  typedef typename duration::rep    rep;
  typedef typename duration::period period;
private:
  duration d_;  // exposition only

public:
  // 20.591.1.1, construct:
  constexpr time_point();  // has value epoch
  constexpr explicit time_point(const duration& d);  // same as time_point() + d
  template <class Duration2>
    constexpr time_point(const time_point<clock, Duration2>& t);

  // 20.591.1.2, observer:
  constexpr duration time_since_epoch() const;

  // 20.591.1.3, arithmetic:
  time_point& operator+=(const duration& d);
  time_point& operator-=(const duration& d);

  // 20.591.1.4, special values:
  static constexpr time_point min();
  static constexpr time_point max();
};
```

1 `Clock` shall meet the Clock requirements (20.591.2).

2 If `Duration` is not an instance of `duration`, the program is ill-formed.

20.591.1.1 `time_point` constructors [time.point.cons]

```
constexpr time_point();
```

1 *Effects:* Constructs an object of type `time_point`, initializing `d_` with `duration::zero()`. Such a `time_point` object represents the epoch.

```
constexpr explicit time_point(const duration& d);
```

2 *Effects:* Constructs an object of type `time_point`, initializing `d_` with `d`. Such a `time_point` object represents the epoch `+ d`.

```
template <class Duration2>
  constexpr time_point(const time_point<clock, Duration2>& t);
```

3 *Remarks:* This constructor shall not participate in overload resolution unless `Duration2` is implicitly convertible to `duration`.

4 *Effects:* Constructs an object of type `time_point`, initializing `d_` with `t.time_since_epoch()`.

20.591.1.2 `time_point` observer [time.point.observer]

```
constexpr duration time_since_epoch() const;
```

1 *Returns:* `d_`.

20.591.1.3 `time_point` arithmetic [time.point.arithmetic]

```
time_point& operator+=(const duration& d);
```

1 *Effects:* `d_ += d`.

2 *Returns:* `*this`.

```
time_point& operator-=(const duration& d);
```

3 *Effects:* `d_ -= d`.

4 *Returns:* `*this`.

20.591.1.4 `time_point` special values [time.point.special]

```
static constexpr time_point min();
```

1 *Returns:* `time_point(duration::min())`.

```
static constexpr time_point max();
```

2 *Returns:* `time_point(duration::max())`.

20.591.1.5 `time_point` non-member arithmetic [time.point.nonmember]

```
template <class Clock, class Duration1, class Rep2, class Period2>
  constexpr time_point<Clock, common_type_t<Duration1, duration<Rep2, Period2>>>
  operator+(const time_point<Clock, Duration1>& lhs, const duration<Rep2, Period2>& rhs);
```

1 *Returns:* `CT(lhs.time_since_epoch() + rhs)`, where `CT` is the type of the return value.

```
template <class Rep1, class Period1, class Clock, class Duration2>
  constexpr time_point<Clock, common_type_t<duration<Rep1, Period1>, Duration2>>
  operator+(const duration<Rep1, Period1>& lhs, const time_point<Clock, Duration2>& rhs);
```

2 *Returns:* `rhs + lhs`.

```
template <class Clock, class Duration1, class Rep2, class Period2>
  constexpr time_point<Clock, common_type_t<Duration1, duration<Rep2, Period2>>>
  operator-(const time_point<Clock, Duration1>& lhs, const duration<Rep2, Period2>& rhs);
```

3 *Returns:* `lhs + (-rhs)`.

```
template <class Clock, class Duration1, class Duration2>
  constexpr common_type_t<Duration1, Duration2>
  operator-(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);
```

4 *Returns:* `lhs.time_since_epoch() - rhs.time_since_epoch()`.

20.591.1.6 `time_point` comparisons [time.point.comparisons]

```
template <class Clock, class Duration1, class Duration2>
  constexpr bool operator==(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);
```

1 *Returns:* `lhs.time_since_epoch() == rhs.time_since_epoch()`.

```
template <class Clock, class Duration1, class Duration2>
  constexpr bool operator!=(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);
```

2 *Returns:* `!(lhs == rhs)`.

```
template <class Clock, class Duration1, class Duration2>
  constexpr bool operator<(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);
```

3 *Returns:* `lhs.time_since_epoch() < rhs.time_since_epoch()`.

```
template <class Clock, class Duration1, class Duration2>
  constexpr bool operator<=(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);
```

4 *Returns:* `!(rhs < lhs)`.

```
template <class Clock, class Duration1, class Duration2>
  constexpr bool operator>(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);
```

5 *Returns:* `rhs < lhs`.

```
template <class Clock, class Duration1, class Duration2>
  constexpr bool operator>=(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);
```

6 *Returns:* `!(lhs < rhs)`.

20.591.1.7 `time_point_cast` [time.point.cast]

```
template <class ToDuration, class Clock, class Duration>
  constexpr time_point<Clock, ToDuration>
  time_point_cast(const time_point<Clock, Duration>& t);
```

1 *Remarks:* This function shall not participate in overload resolution unless `ToDuration` is an instantiation of `duration`.

2 *Returns:* `time_point<Clock, ToDuration>(duration_cast<ToDuration>(t.time_since_epoch()))`.

20.591.2 Clocks [time.clock]

1 The types defined in this subclause shall satisfy the `TrivialClock` requirements (20.578.3).

20.591.2.1 Class `system_clock` [time.clock.system]

1

Objects of class `system_clock` represent wall clock time from the system-wide realtime clock.

```
class system_clock {
public:
  typedef see below                          rep;
  typedef ratio<unspecified , unspecified >   period;
  typedef chrono::duration<rep, period>      duration;
  typedef chrono::time_point<system_clock>   time_point;
  static constexpr bool is_steady =        unspecified;

  static time_point now() noexcept;
```

```
  // Map to C API
  static time_t      to_time_t  (const time_point& t) noexcept;
  static time_point  from_time_t(time_t t) noexcept;
};
```

`typedef unspecified system_clock::rep;`

1 *Requires:* `system_clock::duration::min() < system_clock::duration::zero()` shall be `true`. [*Note:* This implies that `rep` is a signed type. — *end note*]

`static time_t to_time_t(const time_point& t) noexcept;`

2 *Returns:* A `time_t` object that represents the same point in time as `t` when both values are restricted to the coarser of the precisions of `time_t` and `time_point`. It is implementation defined whether values are rounded or truncated to the required precision.

`static time_point from_time_t(time_t t) noexcept;`

3 *Returns:* A `time_point` object that represents the same point in time as `t` when both values are restricted to the coarser of the precisions of `time_t` and `time_point`. It is implementation defined whether values are rounded or truncated to the required precision.

20.595.0.1 Class `steady_clock` [time.clock.steady]

1

Objects of class `steady_clock` represent clocks for which values of `time_point` never decrease as physical time advances and for which values of `time_point` advance at a steady rate relative to real time. That is, the clock may not be adjusted.

```
class steady_clock {
public:
  typedef unspecified                                rep;
  typedef ratio<unspecified , unspecified >          period;
  typedef chrono::duration<rep, period>              duration;
  typedef chrono::time_point<unspecified, duration>  time_point;
  static constexpr bool is_steady =               true;

  static time_point now() noexcept;
};
```

20.600.0.1 Class `high_resolution_clock` [time.clock.hires]

1

Objects of class `high_resolution_clock` represent clocks with the shortest tick period. `high_resolution_clock` may be a synonym for `system_clock` or `steady_clock`.

```
class high_resolution_clock {
public:
  typedef unspecified                                rep;
  typedef ratio<unspecified , unspecified >          period;
  typedef chrono::duration<rep, period>              duration;
  typedef chrono::time_point<unspecified , duration> time_point;
  static constexpr bool is_steady =               unspecified ;

  static time_point now() noexcept;
};
```

20.605.1 Date and time functions [date.time]

1 Table 60 describes the header `<ctime>`.

Table 60 — Header `<ctime>` synopsis

Type	Name(s)			
Macros:	`NULL`	`CLOCKS_PER_SEC`		
Types:	`size_t`	`clock_t`	`time_t`	
Struct:	`tm`			
Functions:				
`asctime`	`clock`	`difftime`	`localtime`	`strftime`
`ctime`	`gmtime`	`mktime`	`time`	

2 The contents are the same as the Standard C library header `<time.h>`.[231] The functions `asctime`, `ctime`, `gmtime`, and `localtime` are not required to avoid data races (17.607.0.1).

See also: ISO C Clause 7.12, Amendment 1 Clause 4.6.4.

20.606 Class template `scoped_allocator_adaptor` [allocator.adaptor]

20.606.1 Header `<scoped_allocator>` synopsis [allocator.adaptor.syn]

```
// scoped allocator adaptor
template <class OuterAlloc, class... InnerAlloc>
  class scoped_allocator_adaptor;
template <class OuterA1, class OuterA2, class... InnerAllocs>
  bool operator==(const scoped_allocator_adaptor<OuterA1, InnerAllocs...>& a,
                  const scoped_allocator_adaptor<OuterA2, InnerAllocs...>& b) noexcept;
template <class OuterA1, class OuterA2, class... InnerAllocs>
  bool operator!=(const scoped_allocator_adaptor<OuterA1, InnerAllocs...>& a,
                  const scoped_allocator_adaptor<OuterA2, InnerAllocs...>& b) noexcept;
```

1

The class template `scoped_allocator_adaptor` is an allocator template that specifies the memory resource (the outer allocator) to be used by a container (as any other allocator does) and also specifies an inner allocator resource to be passed to the constructor of every element within the container. This adaptor is instantiated with one outer and zero or more inner allocator types. If instantiated with only one allocator type, the inner allocator becomes the `scoped_allocator_adaptor` itself, thus using the same allocator resource for the container and every element within the container and, if the elements themselves are containers, each of their elements recursively. If instantiated with more than one allocator, the first allocator is the outer allocator for use by the container, the second allocator is passed to the constructors of the container's elements, and, if the elements themselves are containers, the third allocator is passed to the elements' elements, and so on. If containers are nested to a depth greater than the number of allocators, the last allocator is used repeatedly, as in the single-allocator case, for any remaining recursions. [*Note:* The `scoped_allocator_adaptor` is derived from the outer allocator type so it can be substituted for the outer allocator type in most expressions. — *end note*]

```
namespace std {
  template <class OuterAlloc, class... InnerAllocs>
    class scoped_allocator_adaptor : public OuterAlloc {
  private:
    typedef allocator_traits<OuterAlloc> OuterTraits; // exposition only
```

231) `strftime` supports the C conversion specifiers `C`, `D`, `e`, `F`, `g`, `G`, `h`, `r`, `R`, `t`, `T`, `u`, `V`, and `z`, and the modifiers `E` and `O`.

```
  scoped_allocator_adaptor<InnerAllocs...> inner;    // exposition only
public:
  typedef OuterAlloc outer_allocator_type;
  typedef see below inner_allocator_type;

  typedef typename OuterTraits::value_type value_type;
  typedef typename OuterTraits::size_type size_type;
  typedef typename OuterTraits::difference_type difference_type;
  typedef typename OuterTraits::pointer pointer;
  typedef typename OuterTraits::const_pointer const_pointer;
  typedef typename OuterTraits::void_pointer void_pointer;
  typedef typename OuterTraits::const_void_pointer const_void_pointer;

  typedef see below propagate_on_container_copy_assignment;
  typedef see below propagate_on_container_move_assignment;
  typedef see below propagate_on_container_swap;

  template <class Tp>
    struct rebind {
      typedef scoped_allocator_adaptor<
        OuterTraits::template rebind_alloc<Tp>, InnerAllocs...> other;
    };

  scoped_allocator_adaptor();
  template <class OuterA2>
    scoped_allocator_adaptor(OuterA2&& outerAlloc,
                             const InnerAllocs&... innerAllocs) noexcept;

  scoped_allocator_adaptor(const scoped_allocator_adaptor& other) noexcept;
  scoped_allocator_adaptor(scoped_allocator_adaptor&& other) noexcept;

  template <class OuterA2>
    scoped_allocator_adaptor(
      const scoped_allocator_adaptor<OuterA2, InnerAllocs...>& other) noexcept;
  template <class OuterA2>
    scoped_allocator_adaptor(
      scoped_allocator_adaptor<OuterA2, InnerAllocs...>&& other) noexcept;

  ~scoped_allocator_adaptor();

  inner_allocator_type& inner_allocator() noexcept;
  const inner_allocator_type& inner_allocator() const noexcept;
  outer_allocator_type& outer_allocator() noexcept;
  const outer_allocator_type& outer_allocator() const noexcept;

  pointer allocate(size_type n);
  pointer allocate(size_type n, const_void_pointer hint);
  void deallocate(pointer p, size_type n);
  size_type max_size() const;

  template <class T, class... Args>
    void construct(T* p, Args&&... args);
  template <class T1, class T2, class... Args1, class... Args2>
    void construct(pair<T1, T2>* p, piecewise_construct_t,
                   tuple<Args1...> x, tuple<Args2...> y);
```

```
    template <class T1, class T2>
      void construct(pair<T1, T2>* p);
    template <class T1, class T2, class U, class V>
      void construct(pair<T1, T2>* p, U&& x, V&& y);
    template <class T1, class T2, class U, class V>
      void construct(pair<T1, T2>* p, const pair<U, V>& x);
    template <class T1, class T2, class U, class V>
      void construct(pair<T1, T2>* p, pair<U, V>&& x);

    template <class T>
      void destroy(T* p);

    scoped_allocator_adaptor select_on_container_copy_construction() const;
  };

  template <class OuterA1, class OuterA2, class... InnerAllocs>
    bool operator==(const scoped_allocator_adaptor<OuterA1, InnerAllocs...>& a,
                    const scoped_allocator_adaptor<OuterA2, InnerAllocs...>& b) noexcept;
  template <class OuterA1, class OuterA2, class... InnerAllocs>
    bool operator!=(const scoped_allocator_adaptor<OuterA1, InnerAllocs...>& a,
                    const scoped_allocator_adaptor<OuterA2, InnerAllocs...>& b) noexcept;
}
```

20.608.1 Scoped allocator adaptor member types [allocator.adaptor.types]

`typedef` *see below* `inner_allocator_type;`

1 *Type:* `scoped_allocator_adaptor<OuterAlloc>` if `sizeof...(InnerAllocs)` is zero; otherwise, `scoped_allocator_adaptor<InnerAllocs...>`.

`typedef` *see below* `propagate_on_container_copy_assignment;`

2 *Type:* `true_type` if `allocator_traits<A>::propagate_on_container_copy_assignment::value` is `true` for any `A` in the set of `OuterAlloc` and `InnerAllocs...`; otherwise, `false_type`.

`typedef` *see below* `propagate_on_container_move_assignment;`

3 *Type:* `true_type` if `allocator_traits<A>::propagate_on_container_move_assignment::value` is `true` for any `A` in the set of `OuterAlloc` and `InnerAllocs...`; otherwise, `false_type`.

`typedef` *see below* `propagate_on_container_swap;`

4 *Type:* `true_type` if `allocator_traits<A>::propagate_on_container_swap::value` is `true` for any `A` in the set of `OuterAlloc` and `InnerAllocs...`; otherwise, `false_type`.

20.608.2 Scoped allocator adaptor constructors [allocator.adaptor.cnstr]

`scoped_allocator_adaptor();`

1

Effects: value-initializes the `OuterAlloc` base class and the `inner` allocator object.

```
template <class OuterA2>
  scoped_allocator_adaptor(OuterA2&& outerAlloc,
                           const InnerAllocs&... innerAllocs) noexcept;
```

1 *Requires:* `OuterAlloc` shall be constructible from `OuterA2`.

2 *Effects:* initializes the `OuterAlloc` base class with `std::forward<OuterA2>(outerAlloc)` and `inner` with `innerAllocs...` (hence recursively initializing each allocator within the adaptor with the corresponding allocator from the argument list).

```
scoped_allocator_adaptor(const scoped_allocator_adaptor& other) noexcept;
```

3 *Effects:* initializes each allocator within the adaptor with the corresponding allocator from `other`.

```
scoped_allocator_adaptor(scoped_allocator_adaptor&& other) noexcept;
```

4 *Effects:* move constructs each allocator within the adaptor with the corresponding allocator from `other`.

```
template <class OuterA2>
  scoped_allocator_adaptor(const scoped_allocator_adaptor<OuterA2,
                                                          InnerAllocs...>& other) noexcept;
```

5 *Requires:* `OuterAlloc` shall be constructible from `OuterA2`.

6 *Effects:* initializes each allocator within the adaptor with the corresponding allocator from `other`.

```
template <class OuterA2>
  scoped_allocator_adaptor(scoped_allocator_adaptor<OuterA2,
                                                    InnerAllocs...>&& other) noexcept;
```

7 *Requires:* `OuterAlloc` shall be constructible from `OuterA2`.

8 *Effects:* initializes each allocator within the adaptor with the corresponding allocator rvalue from `other`.

20.609.1 Scoped allocator adaptor members [allocator.adaptor.members]

1 In the `construct` member functions, *OUTERMOST(x)* is `x` if `x` does not have an `outer_allocator()` member function and *OUTERMOST(x.outer_allocator())* otherwise; *OUTERMOST_ALLOC_TRAITS(x)* is `allocator_traits<decltype(`*OUTERMOST*`(x))>`. [*Note:* *OUTERMOST*(x) and *OUTERMOST_ALLOC_TRAITS*(x) are recursive operations. It is incumbent upon the definition of `outer_allocator()` to ensure that the recursion terminates. It will terminate for all instantiations of `scoped_allocator_adaptor`. — *end note*]

```
inner_allocator_type& inner_allocator() noexcept;
const inner_allocator_type& inner_allocator() const noexcept;
```

2 *Returns:* `*this` if `sizeof...(InnerAllocs)` is zero; otherwise, `inner`.

```
outer_allocator_type& outer_allocator() noexcept;
```

3 *Returns:* `static_cast<OuterAlloc&>(*this)`.

```
const outer_allocator_type& outer_allocator() const noexcept;
```

4 *Returns:* `static_cast<const OuterAlloc&>(*this)`.

```
pointer allocate(size_type n);
```

5 *Returns:* `allocator_traits<OuterAlloc>::allocate(outer_allocator(), n)`.

```
pointer allocate(size_type n, const_void_pointer hint);
```

6 *Returns:* `allocator_traits<OuterAlloc>::allocate(outer_allocator(), n, hint)`.

```
void deallocate(pointer p, size_type n) noexcept;
```

7 *Effects:* `allocator_traits<OuterAlloc>::deallocate(outer_allocator(), p, n)`;

```
size_type max_size() const;
```

8 *Returns:* `allocator_traits<OuterAlloc>::max_size(outer_allocator())`.

```
template <class T, class... Args>
  void construct(T* p, Args&&... args);
```

9 *Effects:*

(9.1) — If `uses_allocator<T, inner_allocator_type>::value` is `false` and `is_constructible<T, Args...>::value` is `true`, calls *OUTERMOST_ALLOC_TRAITS*(`*this`)`::construct(`*`OUTERMOST`*`(*this), p, std::forward<Args>(args)...)`.

(9.2) — Otherwise, if `uses_allocator<T, inner_allocator_type>::value` is `true` and `is_constructible<T, allocator_arg_t, inner_allocator_type, Args...>::value` is `true`, calls *OUTERMOST_ALLOC_TRAITS*(`*this`)`::construct(`*`OUTERMOST`*`(*this), p, allocator_arg, inner_allocator(), std::forward<Args>(args)...)`.

(9.3) — Otherwise, if `uses_allocator<T, inner_allocator_type>::value` is `true` and `is_constructible<T, Args..., inner_allocator_type>::value` is `true`, calls *OUTERMOST_ALLOC_TRAITS*(*this):: `construct(`*`OUTERMOST`*`(*this), p, std::forward<Args>(args)..., inner_allocator())`.

(9.4) — Otherwise, the program is ill-formed. [*Note:* An error will result if `uses_allocator` evaluates to true but the specific constructor does not take an allocator. This definition prevents a silent failure to pass an inner allocator to a contained element. — *end note*]

```
template <class T1, class T2, class... Args1, class... Args2>
  void construct(pair<T1, T2>* p,piecewise_construct_t,
                 tuple<Args1...> x, tuple<Args2...> y);
```

10 *Requires:* all of the types in `Args1` and `Args2` shall be `CopyConstructible` (Table 21).

11 *Effects:* Constructs a `tuple` object `xprime` from `x` by the following rules:

(11.1) — If `uses_allocator<T1, inner_allocator_type>::value` is `false` and `is_constructible<T1, Args1...>::value` is `true`, then `xprime` is `x`.

(11.2) — Otherwise, if `uses_allocator<T1, inner_allocator_type>::value` is `true` and `is_constructible<T1, allocator_arg_t, inner_allocator_type, Args1...>::value` is `true`, then `xprime` is `tuple_cat(tuple<allocator_arg_t, inner_allocator_type&>( allocator_arg, inner_allocator()), std::move(x))`.

(11.3) — Otherwise, if `uses_allocator<T1, inner_allocator_type>::value` is `true` and `is_constructible<T1, Args1..., inner_allocator_type>::value` is `true`, then `xprime` is `tuple_cat(std::move(x), tuple<inner_allocator_type&>(inner_allocator()))`.

(11.4) — Otherwise, the program is ill-formed.

and constructs a `tuple` object `yprime` from `y` by the following rules:

(11.5) — If `uses_allocator<T2, inner_allocator_type>::value` is `false` and `is_constructible<T2, Args2...>::value` is `true`, then `yprime` is `y`.

(11.6) — Otherwise, if `uses_allocator<T2, inner_allocator_type>::value` is `true` and `is_constructible<T2, allocator_arg_t, inner_allocator_type, Args2...>::value` is `true`, then `yprime` is `tuple_cat(tuple<allocator_arg_t, inner_allocator_type&>( allocator_arg, inner_allocator()), std::move(y))`.

(11.7) — Otherwise, if `uses_allocator<T2, inner_allocator_type>::value` is `true` and `is_constructible<T2, Args2..., inner_allocator_type>::value` is `true`, then `yprime` is `tuple_cat(std::move(y), tuple<inner_allocator_type&>(inner_allocator()))`.

(11.8) — Otherwise, the program is ill-formed.

then calls *`OUTERMOST_ALLOC_TRAITS`*`(*this)::construct(`*`OUTERMOST`*`(*this), p, piecewise_construct, std::move(xprime), std::move(yprime))`.

```
template <class T1, class T2>
  void construct(pair<T1, T2>* p);
```

12 *Effects:* Equivalent to `this->construct(p, piecewise_construct, tuple<>(), tuple<>())`.

```
template <class T1, class T2, class U, class V>
  void construct(pair<T1, T2>* p, U&& x, V&& y);
```

13 *Effects:* Equivalent to `this->construct(p, piecewise_construct, forward_as_tuple(std::forward<U>(x)), forward_as_tuple(std::forward<V>(y)))`.

```
template <class T1, class T2, class U, class V>
  void construct(pair<T1, T2>* p, const pair<U, V>& x);
```

14 *Effects:* Equivalent to `this->construct(p, piecewise_construct, forward_as_tuple(x.first), forward_as_tuple(x.second))`.

```
template <class T1, class T2, class U, class V>
  void construct(pair<T1, T2>* p, pair<U, V>&& x);
```

15 *Effects:* Equivalent to `this->construct(p, piecewise_construct, forward_as_tuple(std::forward<U>(x.first)), forward_as_tuple(std::forward<V>(x.second)))`.

```
template <class T>
  void destroy(T* p);
```

16 *Effects:* calls *`OUTERMOST_ALLOC_TRAITS`*`(*this)::destroy(`*`OUTERMOST`*`(*this), p)`.

```
scoped_allocator_adaptor select_on_container_copy_construction() const;
```

17 *Returns:* A new scoped_allocator_adaptor object where each allocator `A` in the adaptor is initialized from the result of calling `allocator_traits<A>::select_on_container_copy_construction()` on the corresponding allocator in `*this`.

20.609.2 Scoped allocator operators [scoped.adaptor.operators]

```
template <class OuterA1, class OuterA2, class... InnerAllocs>
  bool operator==(const scoped_allocator_adaptor<OuterA1, InnerAllocs...>& a,
                  const scoped_allocator_adaptor<OuterA2, InnerAllocs...>& b) noexcept;
```

1 *Returns:* `a.outer_allocator() == b.outer_allocator()` if `sizeof...(InnerAllocs)` is zero; otherwise, `a.outer_allocator() == b.outer_allocator() && a.inner_allocator() == b.inner_allocator()`.

```
template <class OuterA1, class OuterA2, class... InnerAllocs>
  bool operator!=(const scoped_allocator_adaptor<OuterA1, InnerAllocs...>& a,
                  const scoped_allocator_adaptor<OuterA2, InnerAllocs...>& b) noexcept;
```

2 *Returns:* `!(a == b)`.

20.610 Class type_index [type.index]

20.610.1 Header <typeindex> synopsis [type.index.synopsis]

```
namespace std {
  class type_index;
  template <class T> struct hash;
  template<> struct hash<type_index>;
}
```

20.610.2 type_index overview [type.index.overview]

```
namespace std {
  class type_index {
  public:
    type_index(const type_info& rhs) noexcept;
    bool operator==(const type_index& rhs) const noexcept;
    bool operator!=(const type_index& rhs) const noexcept;
    bool operator< (const type_index& rhs) const noexcept;
    bool operator<= (const type_index& rhs) const noexcept;
    bool operator> (const type_index& rhs) const noexcept;
    bool operator>= (const type_index& rhs) const noexcept;
    size_t hash_code() const noexcept;
    const char* name() const noexcept;
  private:
    const type_info* target;     // exposition only
    // Note that the use of a pointer here, rather than a reference,
    // means that the default copy/move constructor and assignment
    // operators will be provided and work as expected.
  };
}
```

1 The class `type_index` provides a simple wrapper for `type_info` which can be used as an index type in associative containers (23.894) and in unordered associative containers (23.932).

20.610.3 type_index members [type.index.members]

`type_index(const type_info& rhs) noexcept;`

1 *Effects:* constructs a `type_index` object, the equivalent of `target = &rhs`.

`bool operator==(const type_index& rhs) const noexcept;`

2 *Returns:* `*target == *rhs.target`

`bool operator!=(const type_index& rhs) const noexcept;`

3 *Returns:* `*target != *rhs.target`

`bool operator<(const type_index& rhs) const noexcept;`

4 *Returns:* `target->before(*rhs.target)`

`bool operator<=(const type_index& rhs) const noexcept;`

5 *Returns:* `!rhs.target->before(*target)`

`bool operator>(const type_index& rhs) const noexcept;`

6 *Returns:* `rhs.target->before(*target)`

```
bool operator>=(const type_index& rhs) const noexcept;
```

7 *Returns:* `!target->before(*rhs.target)`

```
size_t hash_code() const noexcept;
```

1 *Returns:* `target->hash_code()`

```
const char* name() const noexcept;
```

2 *Returns:* `target->name()`

20.611.1 Hash support [type.index.hash]

```
template <> struct hash<type_index>;
```

1 The template specialization shall meet the requirements of class template `hash` (20.443.1). For an object `index` of type `type_index`, `hash<type_index>()(index)` shall evaluate to the same result as `index.hash_code()`.

21 Strings library [strings]

21.1 General [strings.general]

1

This Clause describes components for manipulating sequences of any non-array POD (3.2955) type. In this Clause such types are called *char-like types* , and objects of char-like types are called *char-like objects* or simply *characters*.

1 The following subclauses describe a character traits class, a string class, and null-terminated sequence utilities, as summarized in Table 61.

Table 61 — Strings library summary

Subclause		Header(s)
21.7	Character traits	`<string>`
21.26	String classes	`<string>`

21.261	Null-terminated sequence utilities	`<cctype>`
		`<cwctype>`
		`<cstring>`
		`<cwchar>`
		`<cstdlib>`
		`<cuchar>`

21.7 Character traits [char.traits]

1 This subclause defines requirements on classes representing *character traits*, and defines a class template `char_traits<charT>`, along with four specializations, `char_traits<char>`, `char_traits<char16_t>`, `char_traits<char32_t>`, and `char_traits<wchar_t>`, that satisfy those requirements.

2 Most classes specified in Clauses 21.26 and 27 need a set of related types and functions to complete the definition of their semantics. These types and functions are provided as a set of member typedefs and functions in the template parameter 'traits' used by each such template. This subclause defines the semantics of these members.

3 To specialize those templates to generate a string or iostream class to handle a particular character container type `CharT`, that and its related character traits class `Traits` are passed as a pair of parameters to the string or iostream template as parameters `charT` and `traits`. `Traits::char_type` shall be the same as `CharT`.

4 This subclause specifies a struct template, `char_traits<charT>`, and four explicit specializations of it, `char_traits<char>`, `char_traits<char16_t>`, `char_traits<char32_t>`, and `char_traits<wchar_t>`, all of which appear in the header `<string>` and satisfy the requirements below.

21.7.1 Character traits requirements [char.traits.require]

1

In Table 62, `X` denotes a Traits class defining types and functions for the character container type `CharT`; `c` and `d` denote values of type `CharT`; `p` and `q` denote values of type `const CharT*`; `s` denotes a value of type `CharT*`; `n`, `i` and `j` denote values of type `std::size_t`; `e` and `f` denote values of type `X::int_type`; `pos` denotes a value of type `X::pos_type`; `state` denotes a value of type `X::state_type`; and `r` denotes an lvalue of type `CharT`. Operations on Traits shall not throw exceptions.

Table 62 — Character traits requirements

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`X::char_type`	`charT`	(described in 21.17.1)	compile-time
`X::int_type`		(described in 21.17.1)	compile-time
`X::off_type`		(described in 21.17.1)	compile-time
`X::pos_type`		(described in 21.17.1)	compile-time
`X::state_type`		(described in 21.17.1)	compile-time
`X::eq(c,d)`	`bool`	yields: whether `c` is to be treated as equal to `d`.	constant
`X::lt(c,d)`	`bool`	yields: whether `c` is to be treated as less than `d`.	constant
`X::compare(p,q,n)`	`int`	yields: `0` if for each `i` in `[0,n)`, `X::eq(p[i],q[i])` is true; else, a negative value if, for some `j` in `[0,n)`, `X::lt(p[j],q[j])` is true and for each `i` in `[0,j)` `X::eq(p[i],q[i])` is true; else a positive value.	linear
`X::length(p)`	`std::size_t`	yields: the smallest `i` such that `X::eq(p[i],charT())` is true.	linear
`X::find(p,n,c)`	`const X::char_type*`	yields: the smallest `q` in `[p,p+n)` such that `X::eq(*q,c)` is true, zero otherwise.	linear
`X::move(s,p,n)`	`X::char_type*`	for each `i` in `[0,n)`, performs `X::assign(s[i],p[i])`. Copies correctly even where the ranges `[p,p+n)` and `[s,s+n)` overlap. yields: `s`.	linear
`X::copy(s,p,n)`	`X::char_type*`	pre: `p` not in `[s,s+n)`. yields: `s`. for each `i` in `[0,n)`, performs `X::assign(s[i],p[i])`.	linear
`X::assign(r,d)`	(not used)	assigns `r=d`.	constant
`X::assign(s,n,c)`	`X::char_type*`	for each `i` in `[0,n)`, performs `X::assign(s[i],c)`. yields: `s`.	linear
`X::not_eof(e)`	`int_type`	yields: `e` if `X::eq_int_type(e,X::eof())` is false, otherwise a value `f` such that `X::eq_int_type(f,X::eof())` is false.	constant
`X::to_char_type(e)`	`X::char_type`	yields: if for some `c`, `X::eq_int_type(e,X::to_int_type(c))` is true, `c`; else some unspecified value.	constant

Table 62 — Character traits requirements (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`X::to_int_type(c)`	`X::int_type`	yields: some value `e`, constrained by the definitions of `to_char_type` and `eq_int_type`.	constant
`X::eq_int_type(e,f)`	`bool`	yields: for all `c` and `d`, `X::eq(c,d)` is equal to `X::eq_int_type(X::to_int_type(c), X::to_int_type(d))`; otherwise, yields true if `e` and `f` are both copies of `X::eof()`; otherwise, yields false if one of `e` and `f` is a copy of `X::eof()` and the other is not; otherwise the value is unspecified.	constant
`X::eof()`	`X::int_type`	yields: a value `e` such that `X::eq_int_type(e,X::to_int_type(c))` is false for all values `c`.	constant

1 The struct template

```
template<class charT> struct char_traits;
```

shall be provided in the header `<string>` as a basis for explicit specializations.

21.17.1 traits typedefs [char.traits.typedefs]

```
typedef CHAR_T char_type;
```

1 The type `char_type` is used to refer to the character container type in the implementation of the library classes defined in 21.26 and Clause 27.

```
typedef INT_T int_type;
```

2 *Requires:* For a certain character container type `char_type`, a related container type `INT_T` shall be a type or class which can represent all of the valid characters converted from the corresponding `char_type` values, as well as an end-of-file value, `eof()`. The type `int_type` represents a character container type which can hold end-of-file to be used as a return type of the iostream class member functions.[232]

```
typedef implementation-defined off_type;
typedef implementation-defined pos_type;
```

1 *Requires:* Requirements for `off_type` and `pos_type` are described in 27.2.2 and 27.16.

```
typedef STATE_T state_type;
```

232) If `eof()` can be held in `char_type` then some iostreams operations may give surprising results.

2

Requires: `state_type` shall meet the requirements of `CopyAssignable` (Table 23), `CopyConstructible` (Table 21), and `DefaultConstructible` (Table 19) types.

21.21.1 `char_traits` specializations [char.traits.specializations]

```
namespace std {
  template<> struct char_traits<char>;
  template<> struct char_traits<char16_t>;
  template<> struct char_traits<char32_t>;
  template<> struct char_traits<wchar_t>;
}
```

1 The header `<string>` shall define four specializations of the template struct `char_traits`: `char_traits<char>`, `char_traits<char16_t>`, `char_traits<char32_t>`, and `char_traits<wchar_t>`.

2 The requirements for the members of these specializations are given in Clause 21.7.1.

21.21.1.1 `struct char_traits<char>` [char.traits.specializations.char]

```
namespace std {
  template<> struct char_traits<char> {
    typedef char        char_type;
    typedef int         int_type;
    typedef streamoff   off_type;
    typedef streampos   pos_type;
    typedef mbstate_t   state_type;

    static void assign(char_type& c1, const char_type& c2) noexcept;
    static constexpr bool eq(char_type c1, char_type c2) noexcept;
    static constexpr bool lt(char_type c1, char_type c2) noexcept;

    static int compare(const char_type* s1, const char_type* s2, size_t n);
    static size_t length(const char_type* s);
    static const char_type* find(const char_type* s, size_t n,
                 const char_type& a);
    static char_type* move(char_type* s1, const char_type* s2, size_t n);
    static char_type* copy(char_type* s1, const char_type* s2, size_t n);
    static char_type* assign(char_type* s, size_t n, char_type a);

    static constexpr int_type not_eof(int_type c) noexcept;
    static constexpr char_type to_char_type(int_type c) noexcept;
    static constexpr int_type to_int_type(char_type c) noexcept;
    static constexpr bool eq_int_type(int_type c1, int_type c2) noexcept;
    static constexpr int_type eof() noexcept;
  };
}
```

1 The defined types for `int_type`, `pos_type`, `off_type`, and `state_type` shall be `int`, `streampos`, `streamoff`, and `mbstate_t` respectively.

2 The type `streampos` shall be an implementation-defined type that satisfies the requirements for `pos_type` in 27.2.2 and 27.16.

3 The type `streamoff` shall be an implementation-defined type that satisfies the requirements for `off_type` in 27.2.2 and 27.16.

4

The type `mbstate_t` is defined in `<cwchar>` and can represent any of the conversion states that can occur in an implementation-defined set of supported multibyte character encoding rules.

1

The two-argument member `assign` shall be defined identically to the built-in operator `=`. The two-argument members `eq` and `lt` shall be defined identically to the built-in operators `==` and `<` for type `unsigned char`.

1 The member `eof()` shall return `EOF`.

21.24.0.1 `struct char_traits<char16_t>` [char.traits.specializations.char16_t]

```
namespace std {
  template<> struct char_traits<char16_t> {
    typedef char16_t        char_type;
    typedef uint_least16_t  int_type;
    typedef streamoff       off_type;
    typedef u16streampos    pos_type;
    typedef mbstate_t       state_type;

    static void assign(char_type& c1, const char_type& c2) noexcept;
    static constexpr bool eq(char_type c1, char_type c2) noexcept;
    static constexpr bool lt(char_type c1, char_type c2) noexcept;

    static int compare(const char_type* s1, const char_type* s2, size_t n);
    static size_t length(const char_type* s);
    static const char_type* find(const char_type* s, size_t n,
                                 const char_type& a);
    static char_type* move(char_type* s1, const char_type* s2, size_t n);
    static char_type* copy(char_type* s1, const char_type* s2, size_t n);
    static char_type* assign(char_type* s, size_t n, char_type a);

    static constexpr int_type not_eof(int_type c) noexcept;
    static constexpr char_type to_char_type(int_type c) noexcept;
    static constexpr int_type to_int_type(char_type c) noexcept;
    static constexpr bool eq_int_type(int_type c1, int_type c2) noexcept;
    static constexpr int_type eof() noexcept;
  };
}
```

1 The type `u16streampos` shall be an implementation-defined type that satisfies the requirements for pos_type in 27.2.2 and 27.16.

2 The two-argument members `assign`, `eq`, and `lt` shall be defined identically to the built-in operators `=`, `==`, and `<` respectively.

3

The member `eof()` shall return an implementation-defined constant that cannot appear as a valid UTF-16 code unit.

21.25.0.1 `struct char_traits<char32_t>` [char.traits.specializations.char32_t]

```
namespace std {
  template<> struct char_traits<char32_t> {
    typedef char32_t        char_type;
    typedef uint_least32_t  int_type;
    typedef streamoff       off_type;
    typedef u32streampos    pos_type;
    typedef mbstate_t       state_type;
```

```
    static void assign(char_type& c1, const char_type& c2) noexcept;
    static constexpr bool eq(char_type c1, char_type c2) noexcept;
    static constexpr bool lt(char_type c1, char_type c2) noexcept;

    static int compare(const char_type* s1, const char_type* s2, size_t n);
    static size_t length(const char_type* s);
    static const char_type* find(const char_type* s, size_t n,
                 const char_type& a);
    static char_type* move(char_type* s1, const char_type* s2, size_t n);
    static char_type* copy(char_type* s1, const char_type* s2, size_t n);
    static char_type* assign(char_type* s, size_t n, char_type a);

    static constexpr int_type not_eof(int_type c) noexcept;
    static constexpr char_type to_char_type(int_type c) noexcept;
    static constexpr int_type to_int_type(char_type c) noexcept;
    static constexpr bool eq_int_type(int_type c1, int_type c2) noexcept;
    static constexpr int_type eof() noexcept;
  };
}
```

1 The type `u32streampos` shall be an implementation-defined type that satisfies the requirements for pos_type in 27.2.2 and 27.16.

2 The two-argument members `assign`, `eq`, and `lt` shall be defined identically to the built-in operators `=`, `==`, and `<` respectively.

3 The member `eof()` shall return an implementation-defined constant that cannot appear as a Unicode code point.

21.25.0.2 `struct char_traits<wchar_t>` [char.traits.specializations.wchar.t]

```
namespace std {
  template<> struct char_traits<wchar_t> {
    typedef wchar_t      char_type;
    typedef wint_t       int_type;
    typedef streamoff    off_type;
    typedef wstreampos   pos_type;
    typedef mbstate_t    state_type;

    static void assign(char_type& c1, const char_type& c2) noexcept;
    static constexpr bool eq(char_type c1, char_type c2) noexcept;
    static constexpr bool lt(char_type c1, char_type c2) noexcept;

    static int compare(const char_type* s1, const char_type* s2, size_t n);
    static size_t length(const char_type* s);
    static const char_type* find(const char_type* s, size_t n,
                 const char_type& a);
    static char_type* move(char_type* s1, const char_type* s2, size_t n);
    static char_type* copy(char_type* s1, const char_type* s2, size_t n);
    static char_type* assign(char_type* s, size_t n, char_type a);

    static constexpr int_type not_eof(int_type c) noexcept;
    static constexpr char_type to_char_type(int_type c) noexcept;
    static constexpr int_type to_int_type(char_type c) noexcept;
    static constexpr bool eq_int_type(int_type c1, int_type c2) noexcept;
    static constexpr int_type eof() noexcept;
```

```
  };
}
```

1 The defined types for `int_type`, `pos_type`, and `state_type` shall be `wint_t`, `wstreampos`, and `mbstate_t` respectively.

2 The type `wstreampos` shall be an implementation-defined type that satisfies the requirements for pos_type in 27.2.2 and 27.16.

3 The type `mbstate_t` is defined in `<cwchar>` and can represent any of the conversion states that can occur in an implementation-defined set of supported multibyte character encoding rules.

4 The two-argument members `assign`, `eq`, and `lt` shall be defined identically to the built-in operators `=`, `==`, and `<` respectively.

5 The member `eof()` shall return `WEOF`.

21.26 String classes [string.classes]

1

The header `<string>` defines the `basic_string` class template for manipulating varying-length sequences of char-like objects and four typedefs, `string`, `u16string`, `u32string`, and `wstring`, that name the specializations `basic_string<char>`, `basic_string<char16_t>`, `basic_string<char32_t>`, and `basic_string<wchar_t>`, respectively.

Header `<string>` synopsis

```
#include <initializer_list>

namespace std {

  // 21.7, character traits:
  template<class charT> struct char_traits;
  template <> struct char_traits<char>;
  template <> struct char_traits<char16_t>;
  template <> struct char_traits<char32_t>;
  template <> struct char_traits<wchar_t>;

  // 21.28, basic_string:
  template<class charT, class traits = char_traits<charT>,
    class Allocator = allocator<charT> >
      class basic_string;

  template<class charT, class traits, class Allocator>
    basic_string<charT,traits,Allocator>
      operator+(const basic_string<charT,traits,Allocator>& lhs,
                const basic_string<charT,traits,Allocator>& rhs);
  template<class charT, class traits, class Allocator>
    basic_string<charT,traits,Allocator>
      operator+(basic_string<charT,traits,Allocator>&& lhs,
                const basic_string<charT,traits,Allocator>& rhs);
  template<class charT, class traits, class Allocator>
    basic_string<charT,traits,Allocator>
      operator+(const basic_string<charT,traits,Allocator>& lhs,
                basic_string<charT,traits,Allocator>&& rhs);
  template<class charT, class traits, class Allocator>
    basic_string<charT,traits,Allocator>
      operator+(basic_string<charT,traits,Allocator>&& lhs,
```

```
              basic_string<charT,traits,Allocator>&& rhs);
template<class charT, class traits, class Allocator>
  basic_string<charT,traits,Allocator>
    operator+(const charT* lhs,
              const basic_string<charT,traits,Allocator>& rhs);
template<class charT, class traits, class Allocator>
  basic_string<charT,traits,Allocator>
    operator+(const charT* lhs,
              basic_string<charT,traits,Allocator>&& rhs);
template<class charT, class traits, class Allocator>
  basic_string<charT,traits,Allocator>
    operator+(charT lhs, const basic_string<charT,traits,Allocator>& rhs);
template<class charT, class traits, class Allocator>
  basic_string<charT,traits,Allocator>
    operator+(charT lhs, basic_string<charT,traits,Allocator>&& rhs);
template<class charT, class traits, class Allocator>
  basic_string<charT,traits,Allocator>
    operator+(const basic_string<charT,traits,Allocator>& lhs,
              const charT* rhs);
template<class charT, class traits, class Allocator>
  basic_string<charT,traits,Allocator>
    operator+(basic_string<charT,traits,Allocator>&& lhs,
              const charT* rhs);
template<class charT, class traits, class Allocator>
  basic_string<charT,traits,Allocator>
    operator+(const basic_string<charT,traits,Allocator>& lhs, charT rhs);
template<class charT, class traits, class Allocator>
  basic_string<charT,traits,Allocator>
    operator+(basic_string<charT,traits,Allocator>&& lhs, charT rhs);

template<class charT, class traits, class Allocator>
  bool operator==(const basic_string<charT,traits,Allocator>& lhs,
                  const basic_string<charT,traits,Allocator>& rhs) noexcept;
template<class charT, class traits, class Allocator>
  bool operator==(const charT* lhs,
                  const basic_string<charT,traits,Allocator>& rhs);
template<class charT, class traits, class Allocator>
  bool operator==(const basic_string<charT,traits,Allocator>& lhs,
                  const charT* rhs);
template<class charT, class traits, class Allocator>
  bool operator!=(const basic_string<charT,traits,Allocator>& lhs,
                  const basic_string<charT,traits,Allocator>& rhs) noexcept;
template<class charT, class traits, class Allocator>
  bool operator!=(const charT* lhs,
                  const basic_string<charT,traits,Allocator>& rhs);
template<class charT, class traits, class Allocator>
  bool operator!=(const basic_string<charT,traits,Allocator>& lhs,
                  const charT* rhs);

template<class charT, class traits, class Allocator>
  bool operator< (const basic_string<charT,traits,Allocator>& lhs,
                  const basic_string<charT,traits,Allocator>& rhs) noexcept;
template<class charT, class traits, class Allocator>
  bool operator< (const basic_string<charT,traits,Allocator>& lhs,
                  const charT* rhs);
```

```
template<class charT, class traits, class Allocator>
  bool operator< (const charT* lhs,
                  const basic_string<charT,traits,Allocator>& rhs);
template<class charT, class traits, class Allocator>
  bool operator> (const basic_string<charT,traits,Allocator>& lhs,
                  const basic_string<charT,traits,Allocator>& rhs) noexcept;
template<class charT, class traits, class Allocator>
  bool operator> (const basic_string<charT,traits,Allocator>& lhs,
                  const charT* rhs);
template<class charT, class traits, class Allocator>
  bool operator> (const charT* lhs,
                  const basic_string<charT,traits,Allocator>& rhs);

template<class charT, class traits, class Allocator>
  bool operator<=(const basic_string<charT,traits,Allocator>& lhs,
                  const basic_string<charT,traits,Allocator>& rhs) noexcept;
template<class charT, class traits, class Allocator>
  bool operator<=(const basic_string<charT,traits,Allocator>& lhs,
                  const charT* rhs);
template<class charT, class traits, class Allocator>
  bool operator<=(const charT* lhs,
                  const basic_string<charT,traits,Allocator>& rhs);
template<class charT, class traits, class Allocator>
  bool operator>=(const basic_string<charT,traits,Allocator>& lhs,
                  const basic_string<charT,traits,Allocator>& rhs) noexcept;
template<class charT, class traits, class Allocator>
  bool operator>=(const basic_string<charT,traits,Allocator>& lhs,
                  const charT* rhs);
template<class charT, class traits, class Allocator>
  bool operator>=(const charT* lhs,
                  const basic_string<charT,traits,Allocator>& rhs);

// 21.217.1.8, swap:
template<class charT, class traits, class Allocator>
  void swap(basic_string<charT,traits,Allocator>& lhs,
    basic_string<charT,traits,Allocator>& rhs);

// 21.217.1.9, inserters and extractors:
template<class charT, class traits, class Allocator>
  basic_istream<charT,traits>&
    operator>>(basic_istream<charT,traits>& is,
               basic_string<charT,traits,Allocator>& str);
template<class charT, class traits, class Allocator>
  basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os,
               const basic_string<charT,traits,Allocator>& str);
template<class charT, class traits, class Allocator>
  basic_istream<charT,traits>&
    getline(basic_istream<charT,traits>& is,
            basic_string<charT,traits,Allocator>& str,
            charT delim);
template<class charT, class traits, class Allocator>
  basic_istream<charT,traits>&
    getline(basic_istream<charT,traits>&& is,
            basic_string<charT,traits,Allocator>& str,
```

```
            charT delim);
template<class charT, class traits, class Allocator>
  basic_istream<charT,traits>&
    getline(basic_istream<charT,traits>& is,
            basic_string<charT,traits,Allocator>& str);
template<class charT, class traits, class Allocator>
  basic_istream<charT,traits>&
    getline(basic_istream<charT,traits>&& is,
            basic_string<charT,traits,Allocator>& str);

// basic_string typedef names
typedef basic_string<char> string;
typedef basic_string<char16_t> u16string;
typedef basic_string<char32_t> u32string;
typedef basic_string<wchar_t> wstring;

// 21.242, numeric conversions:
int stoi(const string& str, size_t* idx = 0, int base = 10);
long stol(const string& str, size_t* idx = 0, int base = 10);
unsigned long stoul(const string& str, size_t* idx = 0, int base = 10);
long long stoll(const string& str, size_t* idx = 0, int base = 10);
unsigned long long stoull(const string& str, size_t* idx = 0, int base = 10);
float stof(const string& str, size_t* idx = 0);
double stod(const string& str, size_t* idx = 0);
long double stold(const string& str, size_t* idx = 0);
string to_string(int val);
string to_string(unsigned val);
string to_string(long val);
string to_string(unsigned long val);
string to_string(long long val);
string to_string(unsigned long long val);
string to_string(float val);
string to_string(double val);
string to_string(long double val);

int stoi(const wstring& str, size_t* idx = 0, int base = 10);
long stol(const wstring& str, size_t* idx = 0, int base = 10);
unsigned long stoul(const wstring& str, size_t* idx = 0, int base = 10);
long long stoll(const wstring& str, size_t* idx = 0, int base = 10);
unsigned long long stoull(const wstring& str, size_t* idx = 0, int base = 10);
float stof(const wstring& str, size_t* idx = 0);
double stod(const wstring& str, size_t* idx = 0);
long double stold(const wstring& str, size_t* idx = 0);
wstring to_wstring(int val);
wstring to_wstring(unsigned val);
wstring to_wstring(long val);
wstring to_wstring(unsigned long val);
wstring to_wstring(long long val);
wstring to_wstring(unsigned long long val);
wstring to_wstring(float val);
wstring to_wstring(double val);
wstring to_wstring(long double val);

// 21.258, hash support:
template <class T> struct hash;
```

```
  template <> struct hash<string>;
  template <> struct hash<u16string>;
  template <> struct hash<u32string>;
  template <> struct hash<wstring>;

inline namespace literals {
inline namespace string_literals {

  // 21.259, suffix for basic_string literals:
  string    operator "" s(const char* str, size_t len);
  u16string operator "" s(const char16_t* str, size_t len);
  u32string operator "" s(const char32_t* str, size_t len);
  wstring   operator "" s(const wchar_t* str, size_t len);

}
}
}
```

21.28 Class template `basic_string` [basic.string]

1

The class template `basic_string` describes objects that can store a sequence consisting of a varying number of arbitrary char-like objects with the first element of the sequence at position zero. Such a sequence is also called a "string" if the type of the char-like objects that it holds is clear from context. In the rest of this Clause, the type of the char-like objects held in a `basic_string` object is designated by `charT`.

1 The member functions of `basic_string` use an object of the `Allocator` class passed as a template parameter to allocate and free storage for the contained char-like objects.[233]

2

The iterators supported by `basic_string` are random access iterators (24.167.1).

1 In all cases, `size() <= capacity()`.

2 The functions described in this Clause can report two kinds of errors, each associated with an exception type:

(2.1) — a *length* error is associated with exceptions of type `length_error` (19.3.4);

(2.2) — an *out-of-range* error is associated with exceptions of type `out_of_range` (19.4.1).

```
namespace std {
  template<class charT, class traits = char_traits<charT>,
    class Allocator = allocator<charT> >
  class basic_string {
  public:
    // types:
    typedef          traits                                        traits_type;
    typedef typename traits::char_type                             value_type;
    typedef          Allocator                                     allocator_type;
    typedef typename allocator_traits<Allocator>::size_type        size_type;
    typedef typename allocator_traits<Allocator>::difference_type  difference_type;

    typedef value_type& reference;
    typedef const value_type&   const_reference;
```

233) `Allocator::value_type` must name the same type as `charT` (21.31.1).

```
typedef typename allocator_traits<Allocator>::pointer          pointer;
typedef typename allocator_traits<Allocator>::const_pointer    const_pointer;

typedef implementation-defined              iterator;       // See 23.2
typedef implementation-defined              const_iterator; // See 23.2
typedef std::reverse_iterator<iterator> reverse_iterator;
typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
static const size_type npos = -1;

// 21.85.1, construct/copy/destroy:
basic_string() : basic_string(Allocator()) { }
explicit basic_string(const Allocator& a);
basic_string(const basic_string& str);
basic_string(basic_string&& str) noexcept;
basic_string(const basic_string& str, size_type pos, size_type n = npos,
             const Allocator& a = Allocator());
basic_string(const charT* s,
             size_type n, const Allocator& a = Allocator());
basic_string(const charT* s, const Allocator& a = Allocator());
basic_string(size_type n, charT c, const Allocator& a = Allocator());
template<class InputIterator>
  basic_string(InputIterator begin, InputIterator end,
               const Allocator& a = Allocator());
basic_string(initializer_list<charT>, const Allocator& = Allocator());
basic_string(const basic_string&, const Allocator&);
basic_string(basic_string&&, const Allocator&);

~basic_string();
basic_string& operator=(const basic_string& str);
basic_string& operator=(basic_string&& str) noexcept;
basic_string& operator=(const charT* s);
basic_string& operator=(charT c);
basic_string& operator=(initializer_list<charT>);

// 21.121.1, iterators:
iterator       begin() noexcept;
const_iterator begin() const noexcept;
iterator       end() noexcept;
const_iterator end() const noexcept;

reverse_iterator       rbegin() noexcept;
const_reverse_iterator rbegin() const noexcept;
reverse_iterator       rend() noexcept;
const_reverse_iterator rend() const noexcept;

const_iterator         cbegin() const noexcept;
const_iterator         cend() const noexcept;
const_reverse_iterator crbegin() const noexcept;
const_reverse_iterator crend() const noexcept;

// 21.122.1, capacity:
size_type size() const noexcept;
size_type length() const noexcept;
size_type max_size() const noexcept;
void resize(size_type n, charT c);
```

```
void resize(size_type n);
size_type capacity() const noexcept;
void reserve(size_type res_arg = 0);
void shrink_to_fit();
void clear() noexcept;
bool empty() const noexcept;

// 21.141.1, element access:
const_reference operator[](size_type pos) const;
reference       operator[](size_type pos);
const_reference at(size_type n) const;
reference       at(size_type n);

const charT& front() const;
charT& front();
const charT& back() const;
charT& back();

// 21.182.1, modifiers:
basic_string& operator+=(const basic_string& str);
basic_string& operator+=(const charT* s);
basic_string& operator+=(charT c);
basic_string& operator+=(initializer_list<charT>);
basic_string& append(const basic_string& str);
basic_string& append(const basic_string& str, size_type pos,
                     size_type n = npos);
basic_string& append(const charT* s, size_type n);
basic_string& append(const charT* s);
basic_string& append(size_type n, charT c);
template<class InputIterator>
  basic_string& append(InputIterator first, InputIterator last);
basic_string& append(initializer_list<charT>);
void push_back(charT c);

basic_string& assign(const basic_string& str);
basic_string& assign(basic_string&& str) noexcept;
basic_string& assign(const basic_string& str, size_type pos,
                     size_type n = npos);
basic_string& assign(const charT* s, size_type n);
basic_string& assign(const charT* s);
basic_string& assign(size_type n, charT c);
template<class InputIterator>
  basic_string& assign(InputIterator first, InputIterator last);
basic_string& assign(initializer_list<charT>);

basic_string& insert(size_type pos1, const basic_string& str);
basic_string& insert(size_type pos1, const basic_string& str,
                     size_type pos2, size_type n = npos);
basic_string& insert(size_type pos, const charT* s, size_type n);
basic_string& insert(size_type pos, const charT* s);
basic_string& insert(size_type pos, size_type n, charT c);
iterator insert(const_iterator p, charT c);
iterator insert(const_iterator p, size_type n, charT c);
template<class InputIterator>
  iterator insert(const_iterator p, InputIterator first, InputIterator last);
```

```
iterator insert(const_iterator p, initializer_list<charT>);

basic_string& erase(size_type pos = 0, size_type n = npos);
iterator erase(const_iterator p);
iterator erase(const_iterator first, const_iterator last);

void pop_back();

basic_string& replace(size_type pos1, size_type n1,
                      const basic_string& str);
basic_string& replace(size_type pos1, size_type n1,
                      const basic_string& str,
                      size_type pos2, size_type n2 = npos);
basic_string& replace(size_type pos, size_type n1, const charT* s,
                      size_type n2);
basic_string& replace(size_type pos, size_type n1, const charT* s);
basic_string& replace(size_type pos, size_type n1, size_type n2,
                      charT c);

basic_string& replace(const_iterator i1, const_iterator i2,
          const basic_string& str);
basic_string& replace(const_iterator i1, const_iterator i2, const charT* s,
                      size_type n);
basic_string& replace(const_iterator i1, const_iterator i2, const charT* s);
basic_string& replace(const_iterator i1, const_iterator i2,
                      size_type n, charT c);
template<class InputIterator>
  basic_string& replace(const_iterator i1, const_iterator i2,
                        InputIterator j1, InputIterator j2);
basic_string& replace(const_iterator, const_iterator, initializer_list<charT>);

size_type copy(charT* s, size_type n, size_type pos = 0) const;
void swap(basic_string& str);

// 21.197.1, string operations:
const charT* c_str() const noexcept;
const charT* data() const noexcept;
allocator_type get_allocator() const noexcept;

size_type find (const basic_string& str, size_type pos = 0) const noexcept;
size_type find (const charT* s, size_type pos, size_type n) const;
size_type find (const charT* s, size_type pos = 0) const;
size_type find (charT c, size_type pos = 0) const;
size_type rfind(const basic_string& str, size_type pos = npos) const noexcept;
size_type rfind(const charT* s, size_type pos, size_type n) const;
size_type rfind(const charT* s, size_type pos = npos) const;
size_type rfind(charT c, size_type pos = npos) const;

size_type find_first_of(const basic_string& str,
                        size_type pos = 0) const noexcept;
size_type find_first_of(const charT* s,
                        size_type pos, size_type n) const;
size_type find_first_of(const charT* s, size_type pos = 0) const;
size_type find_first_of(charT c, size_type pos = 0) const;
size_type find_last_of (const basic_string& str,
```

```
                          size_type pos = npos) const noexcept;
    size_type find_last_of (const charT* s,
                          size_type pos, size_type n) const;
    size_type find_last_of (const charT* s, size_type pos = npos) const;
    size_type find_last_of (charT c, size_type pos = npos) const;

    size_type find_first_not_of(const basic_string& str,
                size_type pos = 0) const noexcept;
    size_type find_first_not_of(const charT* s, size_type pos,
                              size_type n) const;
    size_type find_first_not_of(const charT* s, size_type pos = 0) const;
    size_type find_first_not_of(charT c, size_type pos = 0) const;
    size_type find_last_not_of (const basic_string& str,
                              size_type pos = npos) const noexcept;
    size_type find_last_not_of (const charT* s, size_type pos,
                              size_type n) const;
    size_type find_last_not_of (const charT* s,
                              size_type pos = npos) const;
    size_type find_last_not_of (charT c, size_type pos = npos) const;

    basic_string substr(size_type pos = 0, size_type n = npos) const;
    int compare(const basic_string& str) const noexcept;
    int compare(size_type pos1, size_type n1,
                const basic_string& str) const;
    int compare(size_type pos1, size_type n1,
                const basic_string& str,
                size_type pos2, size_type n2 = npos) const;
    int compare(const charT* s) const;
    int compare(size_type pos1, size_type n1,
                const charT* s) const;
    int compare(size_type pos1, size_type n1,
                const charT* s, size_type n2) const;
  };
}
```

21.31.1 `basic_string` general requirements [string.require]

1

If any operation would cause `size()` to exceed `max_size()`, that operation shall throw an exception object of type `length_error`.

1 If any member function or operator of `basic_string` throws an exception, that function or operator shall have no other effect.

2 In every specialization `basic_string<charT, traits, Allocator>`, the type `allocator_traits<Allocator>::value_type` shall name the same type as `charT`. Every object of type `basic_string<charT, traits, Allocator>` shall use an object of type `Allocator` to allocate and free storage for the contained `charT` objects as needed. The `Allocator` object used shall be obtained as described in 23.2.1.

3

The char-like objects in a `basic_string` object shall be stored contiguously. That is, for any `basic_string` object `s`, the identity `&*(s.begin() + n) == &*s.begin() + n` shall hold for all values of `n` such that `0 <= n < s.size()`.

1

References, pointers, and iterators referring to the elements of a `basic_string` sequence may be invalidated by the following uses of that `basic_string` object:

— as an argument to any standard library function taking a reference to non-const `basic_string` as an argument.[234]

— Calling non-const member functions, except `operator[]`, `at`, `front`, `back`, `begin`, `rbegin`, `end`, and `rend`.

21.85.1 `basic_string` constructors and assignment operators [string.cons]

```
explicit basic_string(const Allocator& a);
```

1 *Effects:* Constructs an object of class `basic_string`. The postconditions of this function are indicated in Table 63.

Table 63 — `basic_string(const Allocator&)` effects

Element	Value
`data()`	a non-null pointer that is copyable and can have 0 added to it
`size()`	0
`capacity()`	an unspecified value

```
basic_string(const basic_string& str);
basic_string(basic_string&& str) noexcept;
```

1 *Effects:* Constructs an object of class `basic_string` as indicated in Table 64. In the second form, `str` is left in a valid state with an unspecified value.

Table 64 — `basic_string(const basic_string&)` effects

Element	Value
`data()`	points at the first element of an allocated copy of the array whose first element is pointed at by `str.data()`
`size()`	`str.size()`
`capacity()`	a value at least as large as `size()`

```
basic_string(const basic_string& str,
             size_type pos, size_type n = npos,
             const Allocator& a = Allocator());
```

1 *Requires:* `pos <= str.size()`

2 *Throws:* `out_of_range` if `pos > str.size()`.

3 *Effects:* Constructs an object of class `basic_string` and determines the effective length `rlen` of the initial string value as the smaller of `n` and `str.size() - pos`, as indicated in Table 65.

234) For example, as an argument to non-member functions `swap()` (21.217.1.8), `operator>>()` (21.217.1.9), and `getline()` (21.217.1.9), or as an argument to `basic_string::swap()`

Table 65 — `basic_string(const basic_string&, size_type, size_type, const Allocator&)` effects

Element	Value
`data()`	points at the first element of an allocated copy of `rlen` consecutive elements of the string controlled by `str` beginning at position `pos`
`size()`	`rlen`
`capacity()`	a value at least as large as `size()`

```
basic_string(const charT* s, size_type n,
             const Allocator& a = Allocator());
```

1

Requires: `s` points to an array of at least `n` elements of `charT`.

1 *Effects:* Constructs an object of class `basic_string` and determines its initial string value from the array of `charT` of length `n` whose first element is designated by `s`, as indicated in Table 66.

Table 66 — `basic_string(const charT*, size_type, const Allocator&)` effects

Element	Value
`data()`	points at the first element of an allocated copy of the array whose first element is pointed at by `s`
`size()`	`n`
`capacity()`	a value at least as large as `size()`

```
basic_string(const charT* s, const Allocator& a = Allocator());
```

2

Requires: `s` points to an array of at least `traits::length(s) + 1` elements of `charT`.

1

Effects: Constructs an object of class `basic_string` and determines its initial string value from the array of `charT` of length `traits::length(s)` whose first element is designated by `s`, as indicated in Table 67.

Table 67 — `basic_string(const charT*, const Allocator&)` effects

Element	Value
`data()`	points at the first element of an allocated copy of the array whose first element is pointed at by `s`
`size()`	`traits::length(s)`
`capacity()`	a value at least as large as `size()`

1 *Remarks:* Uses `traits::length()`.

```
basic_string(size_type n, charT c, const Allocator& a = Allocator());
```

2 *Requires:* `n < npos`

3 *Effects:* Constructs an object of class `basic_string` and determines its initial string value by repeating the char-like object `c` for all `n` elements, as indicated in Table 68.

Table 68 — `basic_string(size_t, charT, const Allocator&)` effects

Element	Value
`data()`	points at the first element of an allocated array of `n` elements, each storing the initial value `c`
`size()`	`n`
`capacity()`	a value at least as large as `size()`

```
template<class InputIterator>
  basic_string(InputIterator begin, InputIterator end,
               const Allocator& a = Allocator());
```

4 *Effects:* If `InputIterator` is an integral type, equivalent to

```
basic_string(static_cast<size_type>(begin), static_cast<value_type>(end), a)
```

5 Otherwise constructs a string from the values in the range [`begin`, `end`), as indicated in the Sequence Requirements table (see 23.224.1).

```
basic_string(initializer_list<charT> il, const Allocator& a = Allocator());
```

1 *Effects:* Same as `basic_string(il.begin(), il.end(), a)`.

```
basic_string(const basic_string& str, const Allocator& alloc);
basic_string(basic_string&& str, const Allocator& alloc);
```

1 *Effects:* Constructs an object of class `basic_string` as indicated in Table 69. The stored allocator is constructed from `alloc`. In the second form, `str` is left in a valid state with an unspecified value.

Table 69 — `basic_string(const basic_string&, const Allocator&)` and `basic_string(basic_string&&, const Allocator&)` effects

Element	Value
`data()`	points at the first element of an allocated copy of the array whose first element is pointed at by the original value of `str.data()`.
`size()`	the original value of `str.size()`
`capacity()`	a value at least as large as `size()`
`get_allocator()`	`alloc`

1 *Throws:* The second form throws nothing if `alloc == str.get_allocator()`.

```
basic_string& operator=(const basic_string& str);
```

1 *Effects:* If `*this` and `str` are not the same object, modifies `*this` as shown in Table 70.

1 If `*this` and `str` are the same object, the member has no effect.

2 *Returns:* `*this`

Table 70 — `operator=(const basic_string&)` effects

Element	Value
`data()`	points at the first element of an allocated copy of the array whose first element is pointed at by `str.data()`
`size()`	`str.size()`
`capacity()`	a value at least as large as `size()`

```
basic_string& operator=(basic_string&& str) noexcept;
```

3

Effects: If `*this` and `str` are not the same object, modifies `*this` as shown in Table 71. [*Note:* A valid implementation is `swap(str)`. — *end note*]

1 If `*this` and `str` are the same object, the member has no effect.

2 *Returns:* `*this`

Table 71 — `operator=(basic_string&&)` effects

Element	Value
`data()`	points at the array whose first element was pointed at by `str.data()`
`size()`	previous value of `str.size()`
`capacity()`	a value at least as large as `size()`

```
basic_string& operator=(const charT* s);
```

3 *Returns:* `*this = basic_string(s)`.

4 *Remarks:* Uses `traits::length()`.

```
basic_string& operator=(charT c);
```

5 *Returns:* `*this = basic_string(1,c)`.

```
basic_string& operator=(initializer_list<charT> il);
```

6 *Effects:* `*this = basic_string(il)`.

7 *Returns:* `*this`.

21.121.1 `basic_string` iterator support [string.iterators]

```
iterator       begin() noexcept;
const_iterator begin() const noexcept;
const_iterator cbegin() const noexcept;
```

1

Returns: An iterator referring to the first character in the string.

```
iterator       end() noexcept;
const_iterator end() const noexcept;
const_iterator cend() const noexcept;
```

1 *Returns:* An iterator which is the past-the-end value.

```
reverse_iterator       rbegin() noexcept;
const_reverse_iterator rbegin() const noexcept;
const_reverse_iterator crbegin() const noexcept;
```

2 *Returns:* An iterator which is semantically equivalent to `reverse_iterator(end())`.

```
reverse_iterator       rend() noexcept;
const_reverse_iterator rend() const noexcept;
const_reverse_iterator crend() const noexcept;
```

3 *Returns:* An iterator which is semantically equivalent to `reverse_iterator(begin())`.

21.122.1 `basic_string` capacity [string.capacity]

```
size_type size() const noexcept;
```

1

Returns: A count of the number of char-like objects currently in the string.

1

Complexity: Constant time.

```
size_type length() const noexcept;
```

1

Returns: `size()`.

```
size_type max_size() const noexcept;
```

1 *Returns:* The size of the largest possible string.

2 *Complexity:* Constant time.

```
void resize(size_type n, charT c);
```

3 *Requires:* `n <= max_size()`

4 *Throws:* `length_error` if `n > max_size()`.

5

Effects: Alters the length of the string designated by `*this` as follows:

— If `n <= size()`, the function replaces the string designated by `*this` with a string of length `n` whose elements are a copy of the initial elements of the original string designated by `*this`.

— If `n > size()`, the function replaces the string designated by `*this` with a string of length `n` whose first `size()` elements are a copy of the original string designated by `*this`, and whose remaining elements are all initialized to `c`.

```
void resize(size_type n);
```

1 *Effects:* `resize(n,charT())`.

```
size_type capacity() const noexcept;
```

2 *Returns:* The size of the allocated storage in the string.

```
void reserve(size_type res_arg=0);
```

3

The member function `reserve()` is a directive that informs a `basic_string` object of a planned change in size, so that it can manage the storage allocation accordingly.

1

Effects: After `reserve()`, `capacity()` is greater or equal to the argument of `reserve`. [*Note:* Calling `reserve()` with a `res_arg` argument less than `capacity()` is in effect a non-binding shrink request. A call with `res_arg <= size()` is in effect a non-binding shrink-to-fit request. — *end note*]

1 *Throws:* `length_error` if `res_arg > max_size()`.[235]

```
void shrink_to_fit();
```

2 *Remarks:* `shrink_to_fit` is a non-binding request to reduce `capacity()` to `size()`. [*Note:* The request is non-binding to allow latitude for implementation-specific optimizations. — *end note*]

```
void clear() noexcept;
```

3 *Effects:* Behaves as if the function calls:

```
erase(begin(), end());
```

```
bool empty() const noexcept;
```

4 *Returns:* `size() == 0`.

21.141.1 `basic_string` element access [string.access]

```
const_reference operator[](size_type pos) const;
reference       operator[](size_type pos);
```

1 *Requires:* `pos <= size()`.

2

Returns: `*(begin() + pos)` if `pos < size()`. Otherwise, returns a reference to an object of type `charT` with value `charT()`, where modifying the object leads to undefined behavior.

1 *Throws:* Nothing.

2

Complexity: Constant time.

```
const_reference at(size_type pos) const;
reference       at(size_type pos);
```

1 *Throws:* `out_of_range` if `pos >= size()`.

2

Returns: `operator[](pos)`.

```
const charT& front() const;
charT& front();
```

1 *Requires:* `!empty()`

2 *Effects:* Equivalent to `operator[](0)`.

```
const charT& back() const;
charT& back();
```

3 *Requires:* `!empty()`

4 *Effects:* Equivalent to `operator[](size() - 1)`.

235) `reserve()` uses `allocator_traits<Allocator>::allocate()` which may throw an appropriate exception.

21.182.1 basic_string modifiers [string.modifiers]

21.182.1.1 basic_string::operator+= [string::op+=]

```
basic_string&
  operator+=(const basic_string& str);
```

1 *Effects:* Calls `append(str)`.

2 *Returns:* `*this`.

```
basic_string& operator+=(const charT* s);
```

3 *Effects:* Calls `append(s)`.

4 *Returns:* `*this`.

```
basic_string& operator+=(charT c);
```

5 *Effects:* Calls `push_back(c)`;

6 *Returns:* `*this`.

```
basic_string& operator+=(initializer_list<charT> il);
```

7 *Effects:* Calls `append(il)`.

8 *Returns:* `*this`.

21.182.1.2 basic_string::append [string::append]

```
basic_string&
  append(const basic_string& str);
```

1 *Effects:* Calls `append(str.data(), str.size())`.

2 *Returns:* `*this`.

```
basic_string&
  append(const basic_string& str, size_type pos, size_type n = npos);
```

1 *Requires:* `pos <= str.size()`

2 *Throws:* `out_of_range` if `pos > str.size()`.

3 *Effects:* Determines the effective length `rlen` of the string to append as the smaller of `n` and `str.size() - pos` and calls `append(str.data() + pos, rlen)`.

4 *Returns:* `*this`.

```
basic_string&
  append(const charT* s, size_type n);
```

1 *Requires:* `s` points to an array of at least `n` elements of `charT`.

2 *Throws:* `length_error` if `size() + n > max_size()`.

3 *Effects:* The function replaces the string controlled by `*this` with a string of length `size() + n` whose first `size()` elements are a copy of the original string controlled by `*this` and whose remaining elements are a copy of the initial `n` elements of `s`.

1 *Returns:* `*this`.

```
basic_string& append(const charT* s);
```

2 *Requires:* `s` points to an array of at least `traits::length(s) + 1` elements of `charT`.

3 *Effects:* Calls `append(s, traits::length(s))`.

4 *Returns:* `*this`.

```
basic_string& append(size_type n, charT c);
```

5 *Effects:* Equivalent to `append(basic_string(n, c))`.

6 *Returns:* `*this`.

```
template<class InputIterator>
  basic_string& append(InputIterator first, InputIterator last);
```

7 *Requires:* `[first,last)` is a valid range.

8 *Effects:* Equivalent to `append(basic_string(first, last))`.

9 *Returns:* `*this`.

```
basic_string& append(initializer_list<charT> il);
```

10 *Effects:* Calls `append(il.begin(), il.size())`.

11 *Returns:* `*this`.

```
void push_back(charT c);
```

12 *Effects:* Equivalent to `append(static_cast<size_type>(1), c)`.

21.186.0.1 `basic_string::assign` [string::assign]

```
basic_string& assign(const basic_string& str);
```

1 *Effects:* Equivalent to `assign(str, 0, npos)`.

2 *Returns:* `*this`.

```
basic_string& assign(basic_string&& str) noexcept;
```

3 *Effects:* The function replaces the string controlled by `*this` with a string of length `str.size()` whose elements are a copy of the string controlled by `str`. [*Note:* A valid implementation is `swap(str)`. — *end note*]

1 *Returns:* `*this`.

```
basic_string&
  assign(const basic_string& str, size_type pos,
         size_type n = npos);
```

2 *Requires:* `pos <= str.size()`

3 *Throws:* `out_of_range` if `pos > str.size()`.

4 *Effects:* Determines the effective length `rlen` of the string to assign as the smaller of `n` and `str.size() - pos` and calls `assign(str.data() + pos rlen)`.

1 *Returns:* `*this`.

```
basic_string& assign(const charT* s, size_type n);
```

2 *Requires:* `s` points to an array of at least `n` elements of `charT`.

3 *Throws:* `length_error` if `n > max_size()`.

4

Effects: Replaces the string controlled by `*this` with a string of length `n` whose elements are a copy of those pointed to by `s`.

1 *Returns:* `*this`.

```
basic_string& assign(const charT* s);
```

2 *Requires:* `s` points to an array of at least `traits::length(s) + 1` elements of `charT`.

3 *Effects:* Calls `assign(s, traits::length(s))`.

4 *Returns:* `*this`.

```
basic_string& assign(initializer_list<charT> il);
```

5 *Effects:* Calls `assign(il.begin(), il.size())`.

6 `*this`.

```
basic_string& assign(size_type n, charT c);
```

7 *Effects:* Equivalent to `assign(basic_string(n, c))`.

8 *Returns:* `*this`.

```
template<class InputIterator>
  basic_string& assign(InputIterator first, InputIterator last);
```

9 *Effects:* Equivalent to `assign(basic_string(first, last))`.

10 *Returns:* `*this`.

21.189.0.1 `basic_string::insert` [string::insert]

```
basic_string&
  insert(size_type pos1,
         const basic_string& str);
```

1 *Requires:* `pos <= size()`.

2 *Throws:* `out_of_range` if `pos > size()`.

3 *Effects:* Calls `insert(pos, str.data(), str.size())`.

4 *Returns:* `*this`.

```
basic_string&
  insert(size_type pos1,
         const basic_string& str,
         size_type pos2, size_type n = npos);
```

5

Requires: `pos1 <= size()` and `pos2 <= str.size()`

1 *Throws:* `out_of_range` if `pos1 > size()` or `pos2 > str.size()`.

2 *Effects:* Determines the effective length `rlen` of the string to insert as the smaller of `n` and `str.size() - pos2` and calls `insert(pos1, str.data() + pos2, rlen)`.

3 *Returns:* `*this`.

```
basic_string&
  insert(size_type pos, const charT* s, size_type n);
```

4 *Requires:* `s` points to an array of at least `n` elements of `charT` and `pos <= size()`.

5 *Throws:* `out_of_range` if `pos > size()` or `length_error` if `size() + n > max_size()`.

6 *Effects:* Replaces the string controlled by `*this` with a string of length `size() + n` whose first `pos` elements are a copy of the initial elements of the original string controlled by `*this` and whose next `n` elements are a copy of the elements in `s` and whose remaining elements are a copy of the remaining elements of the original string controlled by `*this`.

7 *Returns:* `*this`.

```
basic_string&
  insert(size_type pos, const charT* s);
```

8 *Requires:* `pos <= size()` and `s` points to an array of at least `traits::length(s) + 1` elements of `charT`.

9 *Effects:* Equivalent to `insert(pos, s, traits::length(s))`.

10 *Returns:* `*this`.

```
basic_string&
  insert(size_type pos, size_type n, charT c);
```

11 *Effects:* Equivalent to `insert(pos, basic_string(n, c))`.

12 *Returns:* `*this`.

```
iterator insert(const_iterator p, charT c);
```

13 *Requires:* `p` is a valid iterator on `*this`.

14 *Effects:* inserts a copy of `c` before the character referred to by `p`.

15 *Returns:* An iterator which refers to the copy of the inserted character.

```
iterator insert(const_iterator p, size_type n, charT c);
```

16 *Requires:* `p` is a valid iterator on `*this`.

17 *Effects:* inserts `n` copies of `c` before the character referred to by `p`.

18 *Returns:* An iterator which refers to the copy of the first inserted character, or `p` if `n == 0`.

```
template<class InputIterator>
  iterator insert(const_iterator p, InputIterator first, InputIterator last);
```

19 *Requires:* `p` is a valid iterator on `*this`. `[first,last)` is a valid range.

20 *Effects:* Equivalent to `insert(p - begin(), basic_string(first, last))`.

21 *Returns:* An iterator which refers to the copy of the first inserted character, or `p` if `first == last`.

```
iterator insert(const_iterator p, initializer_list<charT> il);
```

22 *Effects:* `insert(p, il.begin(), il.end())`.

23 *Returns:* An iterator which refers to the copy of the first inserted character, or `p` if `i1` is empty.

21.190.0.1 `basic_string::erase` [string::erase]

```
basic_string& erase(size_type pos = 0, size_type n = npos);
```

1 *Requires:* `pos <= size()`

2 *Throws:* `out_of_range` if `pos > size()`.

3

Effects: Determines the effective length `xlen` of the string to be removed as the smaller of `n` and `size() - pos`.

1

The function then replaces the string controlled by `*this` with a string of length `size() - xlen` whose first `pos` elements are a copy of the initial elements of the original string controlled by `*this`, and whose remaining elements are a copy of the elements of the original string controlled by `*this` beginning at position `pos + xlen`.

1 *Returns:* `*this`.

```
iterator erase(const_iterator p);
```

2 *Throws:* Nothing.

3 *Effects:* removes the character referred to by `p`.

4 *Returns:* An iterator which points to the element immediately following `p` prior to the element being erased. If no such element exists, `end()` is returned.

```
iterator erase(const_iterator first, const_iterator last);
```

5

Requires: `first` and `last` are valid iterators on `*this`, defining a range `[first,last)`.

1 *Throws:* Nothing.

2 *Effects:* removes the characters in the range `[first,last)`.

3 *Returns:* An iterator which points to the element pointed to by `last` prior to the other elements being erased. If no such element exists, `end()` is returned.

```
void pop_back();
```

4 *Requires:* `!empty()`

5 *Throws:* Nothing.

6 *Effects:* Equivalent to `erase(size() - 1, 1)`.

21.193.0.1 `basic_string::replace` [string::replace]

```
basic_string&
  replace(size_type pos1, size_type n1,
          const basic_string& str);
```

1 *Requires:* `pos1 <= size()`.

2 *Throws:* `out_of_range` if `pos1 > size()`.

3 *Effects:* Calls `replace(pos1, n1, str.data(), str.size())`.

4 *Returns:* `*this`.

```
basic_string&
  replace(size_type pos1, size_type n1,
          const basic_string& str,
          size_type pos2, size_type n2 = npos);
```

5 *Requires:* `pos1 <= size()` and `pos2 <= str.size()`.

6 *Throws:* `out_of_range` if `pos1 > size()` or `pos2 > str.size()`.

7 *Effects:* Determines the effective length `rlen` of the string to be inserted as the smaller of `n2` and `str.size() - pos2` and calls `replace(pos1, n1, str.data() + pos2, rlen)`.

8 *Returns:* `*this`.

```
basic_string&
  replace(size_type pos1, size_type n1, const charT* s, size_type n2);
```

9 *Requires:* `pos1 <= size()` and `s` points to an array of at least `n2` elements of `charT`.

10 *Throws:* `out_of_range` if `pos1 > size()` or `length_error` if the length of the resulting string would exceed `max_size()` (see below).

11

Effects: Determines the effective length `xlen` of the string to be removed as the smaller of `n1` and `size() - pos1`. If `size() - xlen >= max_size() - n2` throws `length_error`. Otherwise, the function replaces the string controlled by `*this` with a string of length `size() - xlen + n2` whose first `pos1` elements are a copy of the initial elements of the original string controlled by `*this`, whose next `n2` elements are a copy of the initial `n2` elements of `s`, and whose remaining elements are a copy of the elements of the original string controlled by `*this` beginning at position `pos + xlen`.

1 *Returns:* `*this`.

```
basic_string&
  replace(size_type pos, size_type n, const charT* s);
```

2 *Requires:* `pos <= size()` and `s` points to an array of at least `traits::length(s) + 1` elements of `charT`.

3 *Effects:* Calls `replace(pos, n, s, traits::length(s))`.

4 *Returns:* `*this`.

```
basic_string&
  replace(size_type pos1, size_type n1,
          size_type n2, charT c);
```

5 *Effects:* Equivalent to `replace(pos1, n1, basic_string(n2, c))`.

6 *Returns:* `*this`.

```
basic_string& replace(const_iterator i1, const_iterator i2, const basic_string& str);
```

7 *Requires:* `[begin(),i1)` and `[i1,i2)` are valid ranges.

8 *Effects:* Calls `replace(i1 - begin(), i2 - i1, str)`.

9 *Returns:* `*this`.

```
basic_string&
  replace(const_iterator i1, const_iterator i2, const charT* s, size_type n);
```

10 *Requires:* `[begin(),i1)` and `[i1,i2)` are valid ranges and `s` points to an array of at least `n` elements of `charT`.

11 *Effects:* Calls `replace(i1 - begin(), i2 - i1, s, n)`.

12 *Returns:* `*this`.

```
basic_string& replace(const_iterator i1, const_iterator i2, const charT* s);
```

13 *Requires:* `[begin(),i1)` and `[i1,i2)` are valid ranges and `s` points to an array of at least `traits::length(s) + 1` elements of `charT`.

14 *Effects:* Calls `replace(i1 - begin(), i2 - i1, s, traits::length(s))`.

15 *Returns:* `*this`.

```
basic_string& replace(const_iterator i1, const_iterator i2, size_type n,
                      charT c);
```

16 *Requires:* `[begin(),i1)` and `[i1,i2)` are valid ranges.

17 *Effects:* Calls `replace(i1 - begin(), i2 - i1, basic_string(n, c))`.

18 *Returns:* `*this`.

```
template<class InputIterator>
  basic_string& replace(const_iterator i1, const_iterator i2,
                        InputIterator j1, InputIterator j2);
```

19 *Requires:* `[begin(),i1)`, `[i1,i2)` and `[j1,j2)` are valid ranges.

20 *Effects:* Calls `replace(i1 - begin(), i2 - i1, basic_string(j1, j2))`.

21 *Returns:* `*this`.

```
basic_string& replace(const_iterator i1, const_iterator i2,
                      initializer_list<charT> il);
```

22 *Requires:* `[begin(),i1)` and `[i1,i2)` are valid ranges.

23 *Effects:* Calls `replace(i1 - begin(), i2 - i1, il.begin(), il.size())`.

24 *Returns:* `*this`.

21.196.0.1 `basic_string::copy` [**string::copy**]

```
size_type copy(charT* s, size_type n, size_type pos = 0) const;
```

1 *Requires:* `pos <= size()`

2 *Throws:* `out_of_range` if `pos > size()`.

3 *Effects:* Determines the effective length `rlen` of the string to copy as the smaller of `n` and `size() - pos`. `s` shall designate an array of at least `rlen` elements.

The function then replaces the string designated by `s` with a string of length `rlen` whose elements are a copy of the string controlled by `*this` beginning at position `pos`.

The function does not append a null object to the string designated by `s`.

4 *Returns:* `rlen`.

21.196.0.2 basic_string::swap [string::swap]

```
void swap(basic_string& s);
```

1

Postcondition: `*this` contains the same sequence of characters that was in `s`, `s` contains the same sequence of characters that was in `*this`.

1 *Throws:* Nothing.

2 *Complexity:* Constant time.

21.197.1 basic_string string operations [string.ops]

21.197.1.1 basic_string accessors [string.accessors]

```
const charT* c_str() const noexcept;
const charT* data() const noexcept;
```

1 *Returns:* A pointer `p` such that `p + i == &operator[](i)` for each `i` in `[0,size()]`.

2 *Complexity:* Constant time.

3

Requires: The program shall not alter any of the values stored in the character array.

```
allocator_type get_allocator() const noexcept;
```

1

Returns: A copy of the `Allocator` object used to construct the string or, if that allocator has been replaced, a copy of the most recent replacement.

21.203.0.1 basic_string::find [string::find]

```
size_type find(const basic_string& str,
               size_type pos = 0) const noexcept;
```

1

Effects: Determines the lowest position `xpos`, if possible, such that both of the following conditions obtain:

— `pos <= xpos` and `xpos + str.size() <= size()`;

— `traits::eq(at(xpos+I), str.at(I))` for all elements `I` of the string controlled by `str`.

1

Returns: `xpos` if the function can determine such a value for `xpos`. Otherwise, returns `npos`.

1 *Remarks:* Uses `traits::eq()`.

```
size_type find(const charT* s, size_type pos, size_type n) const;
```

2 *Returns:* `find(basic_string(s,n),pos)`.

```
size_type find(const charT* s, size_type pos = 0) const;
```

3 *Requires:* `s` points to an array of at least `traits::length(s) + 1` elements of `charT`.

4 *Returns:* `find(basic_string(s), pos)`.

```
size_type find(charT c, size_type pos = 0) const;
```

5 *Returns:* `find(basic_string(1,c), pos)`.

21.209.0.1 basic_string::rfind [string::rfind]

```
size_type rfind(const basic_string& str,
                size_type pos = npos) const noexcept;
```

1

Effects: Determines the highest position `xpos`, if possible, such that both of the following conditions obtain:

— `xpos <= pos` and `xpos + str.size() <= size()`;

— `traits::eq(at(xpos+I), str.at(I))` for all elements `I` of the string controlled by `str`.

1 *Returns:* `xpos` if the function can determine such a value for `xpos`. Otherwise, returns `npos`.

2 *Remarks:* Uses `traits::eq()`.

```
size_type rfind(const charT* s, size_type pos, size_type n) const;
```

3

Returns: `rfind(basic_string(s, n), pos)`.

```
size_type rfind(const charT* s, size_type pos = npos) const;
```

1 *Requires:* s points to an array of at least `traits::length(s) + 1` elements of `charT`.

2 *Returns:* `rfind(basic_string(s), pos)`.

```
size_type rfind(charT c, size_type pos = npos) const;
```

3 *Returns:* `rfind(basic_string(1,c),pos)`.

21.211.0.1 basic_string::find_first_of [string::find.first.of]

```
size_type
  find_first_of(const basic_string& str,
                size_type pos = 0) const noexcept;
```

1 *Effects:* Determines the lowest position `xpos`, if possible, such that both of the following conditions obtain:

(1.1) — `pos <= xpos` and `xpos < size()`;

(1.2) — `traits::eq(at(xpos), str.at(I))` for some element `I` of the string controlled by `str`.

2 *Returns:* `xpos` if the function can determine such a value for `xpos`. Otherwise, returns `npos`.

3 *Remarks:* Uses `traits::eq()`.

```
size_type
  find_first_of(const charT* s, size_type pos, size_type n) const;
```

4 *Returns:* `find_first_of(basic_string(s, n), pos)`.

```
size_type find_first_of(const charT* s, size_type pos = 0) const;
```

5 *Requires:* `s` points to an array of at least `traits::length(s) + 1` elements of `charT`.

6 *Returns:* `find_first_of(basic_string(s), pos)`.

```
size_type find_first_of(charT c, size_type pos = 0) const;
```

7 *Returns:* `find_first_of(basic_string(1,c), pos)`.

21.211.0.2 basic_string::find_last_of [string::find.last.of]

```
size_type
  find_last_of(const basic_string& str,
               size_type pos = npos) const noexcept;
```

1 *Effects:* Determines the highest position `xpos`, if possible, such that both of the following conditions obtain:

(1.1) — `xpos <= pos` and `xpos < size()`;

(1.2) — `traits::eq(at(xpos), str.at(I))` for some element `I` of the string controlled by `str`.

2 *Returns:* `xpos` if the function can determine such a value for `xpos`. Otherwise, returns `npos`.

3 *Remarks:* Uses `traits::eq()`.

```
size_type find_last_of(const charT* s, size_type pos, size_type n) const;
```

4 *Returns:* `find_last_of(basic_string(s, n), pos)`.

```
size_type find_last_of(const charT* s, size_type pos = npos) const;
```

5 *Requires:* `s` points to an array of at least `traits::length(s) + 1` elements of `charT`.

6 *Returns:* `find_last_of(basic_string(s), pos)`.

```
size_type find_last_of(charT c, size_type pos = npos) const;
```

7 *Returns:* `find_last_of(basic_string(1,c),pos)`.

21.211.0.3 basic_string::find_first_not_of [string::find.first.not.of]

```
size_type
  find_first_not_of(const basic_string& str,
                    size_type pos = 0) const noexcept;
```

1 *Effects:* Determines the lowest position `xpos`, if possible, such that both of the following conditions obtain:

(1.1) — `pos <= xpos` and `xpos < size()`;

(1.2) — `traits::eq(at(xpos), str.at(I))` for no element `I` of the string controlled by `str`.

2 *Returns:* `xpos` if the function can determine such a value for `xpos`. Otherwise, returns `npos`.

3 *Remarks:* Uses `traits::eq()`.

```
size_type
  find_first_not_of(const charT* s, size_type pos, size_type n) const;
```

4 *Returns:* `find_first_not_of(basic_string(s, n), pos)`.

```
size_type find_first_not_of(const charT* s, size_type pos = 0) const;
```

5 *Requires:* `s` points to an array of at least `traits::length(s) + 1` elements of `charT`.

6 *Returns:* `find_first_not_of(basic_string(s), pos)`.

```
size_type find_first_not_of(charT c, size_type pos = 0) const;
```

7 *Returns:* `find_first_not_of(basic_string(1, c), pos)`.

21.211.0.4 basic_string::find_last_not_of [string::find.last.not.of]

```
size_type
  find_last_not_of(const basic_string& str,
                   size_type pos = npos) const noexcept;
```

1 *Effects:* Determines the highest position `xpos`, if possible, such that both of the following conditions obtain:

(1.1) — `xpos <= pos` and `xpos < size()`;

(1.2) — `traits::eq(at(xpos), str.at(I))` for no element `I` of the string controlled by `str`.

2 *Returns:* `xpos` if the function can determine such a value for `xpos`. Otherwise, returns `npos`.

3 *Remarks:* Uses `traits::eq()`.

```
size_type find_last_not_of(const charT* s, size_type pos,
                           size_type n) const;
```

4 *Returns:* `find_last_not_of(basic_string(s, n), pos)`.

```
size_type find_last_not_of(const charT* s, size_type pos = npos) const;
```

5 *Requires:* `s` points to an array of at least `traits::length(s) + 1` elements of `charT`.

6 *Returns:* `find_last_not_of(basic_string(s), pos)`.

```
size_type find_last_not_of(charT c, size_type pos = npos) const;
```

7 *Returns:* `find_last_not_of(basic_string(1, c), pos)`.

21.211.0.5 basic_string::substr [string::substr]

```
basic_string substr(size_type pos = 0, size_type n = npos) const;
```

1 *Requires:* `pos <= size()`

2 *Throws:* `out_of_range` if `pos > size()`.

3 *Effects:* Determines the effective length `rlen` of the string to copy as the smaller of `n` and `size() - pos`.

4 *Returns:* `basic_string(data()+pos,rlen)`.

21.211.0.6 basic_string::compare [string::compare]

```
int compare(const basic_string& str) const noexcept;
```

1

Effects: Determines the effective length *rlen* of the strings to compare as the smallest of `size()` and `str.size()`. The function then compares the two strings by calling `traits::compare(data(), str.data(), rlen)`.

1

Returns: The nonzero result if the result of the comparison is nonzero. Otherwise, returns a value as indicated in Table 72.

```
int compare(size_type pos1, size_type n1,
            const basic_string& str) const;
```

1

Returns: `basic_string(*this,pos1,n1).compare(str)`.

Table 72 — `compare()` results

Condition	Return Value
`size() < str.size()`	`< 0`
`size() == str.size()`	`0`
`size() > str.size()`	`> 0`

```
int compare(size_type pos1, size_type n1,
            const basic_string& str,
            size_type pos2, size_type n2 = npos) const;
```

1 *Returns:* `basic_string(*this, pos1, n1).compare(basic_string(str, pos2, n2))`.

```
int compare(const charT* s) const;
```

2 *Returns:* `compare(basic_string(s))`.

```
int compare(size_type pos, size_type n1,
            const charT* s) const;
```

3 *Returns:* `basic_string(*this, pos, n1).compare(basic_string(s))`.

```
int compare(size_type pos, size_type n1,
            const charT* s, size_type n2) const;
```

4 *Returns:* `basic_string(*this, pos, n1).compare(basic_string(s, n2))`.

21.217.1 basic_string non-member functions [string.nonmembers]

21.217.1.1 operator+ [string::op+]

```
template<class charT, class traits, class Allocator>
  basic_string<charT,traits,Allocator>
    operator+(const basic_string<charT,traits,Allocator>& lhs,
              const basic_string<charT,traits,Allocator>& rhs);
```

1 *Returns:* `basic_string<charT,traits,Allocator>(lhs).append(rhs)`

```
template<class charT, class traits, class Allocator>
  basic_string<charT,traits,Allocator>
    operator+(basic_string<charT,traits,Allocator>&& lhs,
              const basic_string<charT,traits,Allocator>& rhs);
```

2 *Returns:* `std::move(lhs.append(rhs))`

```
template<class charT, class traits, class Allocator>
  basic_string<charT,traits,Allocator>
    operator+(const basic_string<charT,traits,Allocator>& lhs,
              basic_string<charT,traits,Allocator>&& rhs);
```

3 *Returns:* `std::move(rhs.insert(0, lhs))`

```
template<class charT, class traits, class Allocator>
  basic_string<charT,traits,Allocator>
    operator+(basic_string<charT,traits,Allocator>&& lhs,
              basic_string<charT,traits,Allocator>&& rhs);
```

4 *Returns:* `std::move(lhs.append(rhs))` [*Note:* Or equivalently `std::move(rhs.insert(0, lhs))` — *end note*]

```
template<class charT, class traits, class Allocator>
  basic_string<charT,traits,Allocator>
    operator+(const charT* lhs,
              const basic_string<charT,traits,Allocator>& rhs);
```

5 *Returns:* `basic_string<charT,traits,Allocator>(lhs) + rhs`.

6 *Remarks:* Uses `traits::length()`.

```
template<class charT, class traits, class Allocator>
  basic_string<charT,traits,Allocator>
    operator+(const charT* lhs,
              basic_string<charT,traits,Allocator>&& rhs);
```

7 *Returns:* `std::move(rhs.insert(0, lhs))`.

8 *Remarks:* Uses `traits::length()`.

```
template<class charT, class traits, class Allocator>
  basic_string<charT,traits,Allocator>
    operator+(charT lhs,
              const basic_string<charT,traits,Allocator>& rhs);
```

9 *Returns:* `basic_string<charT,traits,Allocator>(1,lhs) + rhs`.

```
template<class charT, class traits, class Allocator>
  basic_string<charT,traits,Allocator>
    operator+(charT lhs,
              basic_string<charT,traits,Allocator>&& rhs);
```

10 *Returns:* `std::move(rhs.insert(0, 1, lhs))`.

```
template<class charT, class traits, class Allocator>
  basic_string<charT,traits,Allocator>
    operator+(const basic_string<charT,traits,Allocator>& lhs,
              const charT* rhs);
```

11 *Returns:* `lhs + basic_string<charT,traits,Allocator>(rhs)`.

12 *Remarks:* Uses `traits::length()`.

```
template<class charT, class traits, class Allocator>
  basic_string<charT,traits,Allocator>
    operator+(basic_string<charT,traits,Allocator>&& lhs,
              const charT* rhs);
```

13 *Returns:* `std::move(lhs.append(rhs))`.

14 *Remarks:* Uses `traits::length()`.

```
template<class charT, class traits, class Allocator>
  basic_string<charT,traits,Allocator>
    operator+(const basic_string<charT,traits,Allocator>& lhs,
              charT rhs);
```

15 *Returns:* `lhs + basic_string<charT,traits,Allocator>(1,rhs)`.

```
template<class charT, class traits, class Allocator>
  basic_string<charT,traits,Allocator>
    operator+(basic_string<charT,traits,Allocator>&& lhs,
              charT rhs);
```

16 *Returns:* `std::move(lhs.append(1, rhs))`.

21.217.1.2 `operator==` [string::operator==]

```
template<class charT, class traits, class Allocator>
  bool operator==(const basic_string<charT,traits,Allocator>& lhs,
                  const basic_string<charT,traits,Allocator>& rhs) noexcept;
```

1 *Returns:* `lhs.compare(rhs) == 0`.

```
template<class charT, class traits, class Allocator>
  bool operator==(const charT* lhs,
                  const basic_string<charT,traits,Allocator>& rhs);
```

2 *Returns:* `rhs == lhs`.

```
template<class charT, class traits, class Allocator>
  bool operator==(const basic_string<charT,traits,Allocator>& lhs,
                  const charT* rhs);
```

3 *Requires:* `rhs` points to an array of at least `traits::length(rhs) + 1` elements of `charT`.

4 *Returns:* `lhs.compare(rhs) == 0`.

21.217.1.3 `operator!=` [string::op!=]

```
template<class charT, class traits, class Allocator>
  bool operator!=(const basic_string<charT,traits,Allocator>& lhs,
                  const basic_string<charT,traits,Allocator>& rhs) noexcept;
```

1 *Returns:* `!(lhs == rhs)`.

```
template<class charT, class traits, class Allocator>
  bool operator!=(const charT* lhs,
                  const basic_string<charT,traits,Allocator>& rhs);
```

2 *Returns:* `rhs != lhs`.

```
template<class charT, class traits, class Allocator>
  bool operator!=(const basic_string<charT,traits,Allocator>& lhs,
                  const charT* rhs);
```

3 *Requires:* `rhs` points to an array of at least `traits::length(rhs) + 1` elements of `charT`.

4 *Returns:* `lhs.compare(rhs) != 0`.

21.217.1.4 `operator<` [string::op<]

```
template<class charT, class traits, class Allocator>
  bool operator< (const basic_string<charT,traits,Allocator>& lhs,
                  const basic_string<charT,traits,Allocator>& rhs) noexcept;
```

1 *Returns:* `lhs.compare(rhs) < 0`.

```
template<class charT, class traits, class Allocator>
  bool operator< (const charT* lhs,
                  const basic_string<charT,traits,Allocator>& rhs);
```

2 *Returns:* `rhs.compare(lhs) > 0`.

```
template<class charT, class traits, class Allocator>
  bool operator< (const basic_string<charT,traits,Allocator>& lhs,
                  const charT* rhs);
```

3 *Returns:* `lhs.compare(rhs) < 0`.

21.217.1.5 operator> [string::op>]

```
template<class charT, class traits, class Allocator>
  bool operator> (const basic_string<charT,traits,Allocator>& lhs,
                  const basic_string<charT,traits,Allocator>& rhs) noexcept;
```

1 *Returns:* `lhs.compare(rhs) > 0`.

```
template<class charT, class traits, class Allocator>
  bool operator> (const charT* lhs,
                  const basic_string<charT,traits,Allocator>& rhs);
```

2 *Returns:* `rhs.compare(lhs) < 0`.

```
template<class charT, class traits, class Allocator>
  bool operator> (const basic_string<charT,traits,Allocator>& lhs,
                  const charT* rhs);
```

3 *Returns:* `lhs.compare(rhs) > 0`.

21.217.1.6 operator<= [string::op<=]

```
template<class charT, class traits, class Allocator>
  bool operator<=(const basic_string<charT,traits,Allocator>& lhs,
                  const basic_string<charT,traits,Allocator>& rhs) noexcept;
```

1 *Returns:* `lhs.compare(rhs) <= 0`.

```
template<class charT, class traits, class Allocator>
  bool operator<=(const charT* lhs,
                  const basic_string<charT,traits,Allocator>& rhs);
```

2 *Returns:* `rhs.compare(lhs) >= 0`.

```
template<class charT, class traits, class Allocator>
  bool operator<=(const basic_string<charT,traits,Allocator>& lhs,
                  const charT* rhs);
```

3 *Returns:* `lhs.compare(rhs) <= 0`.

21.217.1.7 operator>= [string::op>=]

```
template<class charT, class traits, class Allocator>
  bool operator>=(const basic_string<charT,traits,Allocator>& lhs,
                  const basic_string<charT,traits,Allocator>& rhs) noexcept;
```

1 *Returns:* `lhs.compare(rhs) >= 0`.

```
template<class charT, class traits, class Allocator>
  bool operator>=(const charT* lhs,
                  const basic_string<charT,traits,Allocator>& rhs);
```

2 *Returns:* `rhs.compare(lhs) <= 0`.

```
template<class charT, class traits, class Allocator>
  bool operator>=(const basic_string<charT,traits,Allocator>& lhs,
                  const charT* rhs);
```

3 *Returns:* `lhs.compare(rhs) >= 0`.

21.217.1.8 swap [string.special]

```
template<class charT, class traits, class Allocator>
  void swap(basic_string<charT,traits,Allocator>& lhs,
    basic_string<charT,traits,Allocator>& rhs);
```

1 *Effects:* Equivalent to `lhs.swap(rhs);`

21.217.1.9 Inserters and extractors [string.io]

```
template<class charT, class traits, class Allocator>
  basic_istream<charT,traits>&
    operator>>(basic_istream<charT,traits>& is,
               basic_string<charT,traits,Allocator>& str);
```

1

Effects: Behaves as a formatted input function (27.128.0.1.1). After constructing a `sentry` object, if the sentry converts to true, calls `str.erase()` and then extracts characters from `is` and appends them to `str` as if by calling `str.append(1,c)`. If `is.width()` is greater than zero, the maximum number `n` of characters appended is `is.width()`; otherwise `n` is `str.max_size()`. Characters are extracted and appended until any of the following occurs:

— n characters are stored;

— end-of-file occurs on the input sequence;

— `isspace(c,is.getloc())` is true for the next available input character c.

1

After the last character (if any) is extracted, `is.width(0)` is called and the `sentry` object `k` is destroyed.

1

If the function extracts no characters, it calls `is.setstate(ios::failbit)`, which may throw `ios_base::failure` (27.84.0.1).

1 *Returns:* `is`

```
template<class charT, class traits, class Allocator>
  basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os,
               const basic_string<charT,traits,Allocator>& str);
```

2 *Effects:* Behaves as a formatted output function (27.198.0.1.1) of `os`. Forms a character sequence `seq`, initially consisting of the elements defined by the range [`str.begin(), str.end()`). Determines padding for `seq` as described in 27.198.0.1.1. Then inserts `seq` as if by calling `os.rdbuf()->sputn(seq, n)`, where `n` is the larger of `os.width()` and `str.size()`; then calls `os.width(0)`.

3 *Returns:* `os`

```
template<class charT, class traits, class Allocator>
  basic_istream<charT,traits>&
    getline(basic_istream<charT,traits>& is,
            basic_string<charT,traits,Allocator>& str,
            charT delim);
template<class charT, class traits, class Allocator>
  basic_istream<charT,traits>&
    getline(basic_istream<charT,traits>&& is,
            basic_string<charT,traits,Allocator>& str,
            charT delim);
```

4

Effects: Behaves as an unformatted input function (27.154.0.1), except that it does not affect the value returned by subsequent calls to `basic_istream<>::gcount()`. After constructing a `sentry` object, if the sentry converts to true, calls `str.erase()` and then extracts characters from `is` and appends them to `str` as if by calling `str.append(1, c)` until any of the following occurs:

— end-of-file occurs on the input sequence (in which case, the `getline` function calls `is.setstate(ios_base::eofbit)`).

— `traits::eq(c, delim)` for the next available input character *c* (in which case, *c* is extracted but not appended) (27.84.0.1)

— `str.max_size()` characters are stored (in which case, the function calls `is.setstate(ios_base::failbit))` (27.84.0.1)

1

The conditions are tested in the order shown. In any case, after the last character is extracted, the `sentry` object `k` is destroyed.

1 If the function extracts no characters, it calls `is.setstate(ios_base::failbit)` which may throw `ios_base::failure` (27.84.0.1).

2 *Returns:* `is`.

```
template<class charT, class traits, class Allocator>
  basic_istream<charT,traits>&
    getline(basic_istream<charT,traits>& is,
            basic_string<charT,traits,Allocator>& str);
template<class charT, class traits, class Allocator>
  basic_istream<charT,traits>&
    getline(basic_istream<charT,traits>&& is,
            basic_string<charT,traits,Allocator>& str);
```

3 *Returns:* `getline(is,str,is.widen('\n'))`

21.242 Numeric conversions [string.conversions]

```
int stoi(const string& str, size_t* idx = 0, int base = 10);
long stol(const string& str, size_t* idx = 0, int base = 10);
unsigned long stoul(const string& str, size_t* idx = 0, int base = 10);
long long stoll(const string& str, size_t* idx = 0, int base = 10);
unsigned long long stoull(const string& str, size_t* idx = 0, int base = 10);
```

1

Effects: the first two functions call `strtol(str.c_str(), ptr, base)`, and the last three functions call `strtoul(str.c_str(), ptr, base)`, `strtoll(str.c_str(), ptr, base)`, and `strtoull(str.c_str(), ptr, base)`, respectively. Each function returns the converted result, if any. The argument `ptr` designates a pointer to an object internal to the function that is used to determine what to store at `*idx`. If the function does not throw an exception and `idx != 0`, the function stores in `*idx` the index of the first unconverted element of `str`.

1 *Returns:* The converted result.

2

Throws: `invalid_argument` if `strtol`, `strtoul`, `strtoll`, or `strtoull` reports that no conversion could be performed. Throws `out_of_range` if `strtol`, `strtoul`, `strtoll` or `strtoull` sets `errno` to `ERANGE`, or if the converted value is outside the range of representable values for the return type.

```
float stof(const string& str, size_t* idx = 0);
double stod(const string& str, size_t* idx = 0);
long double stold(const string& str, size_t* idx = 0);
```

1

Effects: the first two functions call `strtod(str.c_str(), ptr)` and the third function calls `strtold(str.c_str(), ptr)`. Each function returns the converted result, if any. The argument `ptr` designates a pointer to an object internal to the function that is used to determine what to store at `*idx`. If the function does not throw an exception and `idx != 0`, the function stores in `*idx` the index of the first unconverted element of `str`.

1 *Returns:* The converted result.

2

Throws: `invalid_argument` if `strtod` or `strtold` reports that no conversion could be performed. Throws `out_of_range` if `strtod` or `strtold` sets `errno` to `ERANGE` or if the converted value is outside the range of representable values for the return type.

```
string to_string(int val);
string to_string(unsigned val);
string to_string(long val);
string to_string(unsigned long val);
string to_string(long long val);
string to_string(unsigned long long val);
string to_string(float val);
string to_string(double val);
string to_string(long double val);
```

1

Returns: Each function returns a `string` object holding the character representation of the value of its argument that would be generated by calling `sprintf(buf, fmt, val)` with a format specifier of `"%d"`, `"%u"`, `"%ld"`, `"%lu"`, `"%lld"`, `"%llu"`, `"%f"`, `"%f"`, or `"%Lf"`, respectively, where `buf` designates an internal character buffer of sufficient size.

```
int stoi(const wstring& str, size_t* idx = 0, int base = 10);
long stol(const wstring& str, size_t* idx = 0, int base = 10);
unsigned long stoul(const wstring& str, size_t* idx = 0, int base = 10);
long long stoll(const wstring& str, size_t* idx = 0, int base = 10);
unsigned long long stoull(const wstring& str, size_t* idx = 0, int base = 10);
```

1 *Effects:* the first two functions call `wcstol(str.c_str(), ptr, base)`, and the last three functions call `wcstoul(str.c_str(), ptr, base)`, `wcstoll(str.c_str(), ptr, base)`, and `wcstoull(str.c_str(), ptr, base)`, respectively. Each function returns the converted result, if any. The argument `ptr` designates a pointer to an object internal to the function that is used to determine what to store at `*idx`. If the function does not throw an exception and `idx != 0`, the function stores in `*idx` the index of the first unconverted element of `str`.

2 *Returns:* The converted result.

3 *Throws:* `invalid_argument` if `wcstol`, `wcstoul`, `wcstoll`, or `wcstoull` reports that no conversion could be performed. Throws `out_of_range` if the converted value is outside the range of representable values for the return type.

```
float stof(const wstring& str, size_t* idx = 0);
double stod(const wstring& str, size_t* idx = 0);
long double stold(const wstring& str, size_t* idx = 0);
```

4 *Effects:* the first two functions call `wcstod(str.c_str(), ptr)` and the third function calls `wcstold(str.c_str(), ptr)`. Each function returns the converted result, if any. The argument `ptr` designates a pointer to an object internal to the function that is used to determine what to store at `*idx`. If the function does not throw an exception and `idx != 0`, the function stores in `*idx` the index of the first unconverted element of `str`.

5 *Returns:* The converted result.

6 *Throws:* `invalid_argument` if `wcstod` or `wcstold` reports that no conversion could be performed. Throws `out_of_range` if `wcstod` or `wcstold` sets `errno` to `ERANGE`.

```
wstring to_wstring(int val);
wstring to_wstring(unsigned val);
wstring to_wstring(long val);
wstring to_wstring(unsigned long val);
wstring to_wstring(long long val);
wstring to_wstring(unsigned long long val);
wstring to_wstring(float val);
wstring to_wstring(double val);
wstring to_wstring(long double val);
```

7 *Returns:* Each function returns a `wstring` object holding the character representation of the value of its argument that would be generated by calling `swprintf(buf, buffsz, fmt, val)` with a format specifier of `L"%d"`, `L"%u"`, `L"%ld"`, `L"%lu"`, `L"%lld"`, `L"%llu"`, `L"%f"`, `L"%f"`, or `L"%Lf"`, respectively, where `buf` designates an internal character buffer of sufficient size `buffsz`.

21.258 Hash support [basic.string.hash]

```
template <> struct hash<string>;
template <> struct hash<u16string>;
template <> struct hash<u32string>;
template <> struct hash<wstring>;
```

1 The template specializations shall meet the requirements of class template `hash` (20.443.1).

21.259 Suffix for `basic_string` literals [basic.string.literals]

```
string operator "" s(const char* str, size_t len);
```

1 *Returns:* `string{str,len}`

```
u16string operator "" s(const char16_t* str, size_t len);
```

1 *Returns:* `u16string{str,len}`

```
u32string operator "" s(const char32_t* str, size_t len);
```

2 *Returns:* `u32string{str,len}`

```
wstring operator "" s(const wchar_t* str, size_t len);
```

3 *Returns:* `wstring{str,len}`

4 [*Note:* The same suffix `s` is used for `chrono::duration` literals denoting seconds but there is no conflict, since duration suffixes apply to numbers and string literal suffixes apply to character array literals. — *end note*]

21.261 Null-terminated sequence utilities [c.strings]

1

Tables 74, 75, 76, 77, 78, and 79 describe headers `<cctype>`, `<cwctype>`, `<cstring>`, `<cwchar>`, `<cstdlib>` (character conversions), and `<cuchar>`, respectively.

1

The contents of these headers shall be the same as the Standard C Library headers `<ctype.h>`, `<wctype.h>`, `<string.h>`, `<wchar.h>`, and `<stdlib.h>` and the C Unicode TR header `<uchar.h>`, respectively, with the following modifications:

1

The headers shall not define the types `char16_t`, `char32_t`, and `wchar_t` (2.287).

1 The function signature `strchr(const char*, int)` shall be replaced by the two declarations:

```
const char* strchr(const char* s, int c);
      char* strchr(      char* s, int c);
```

both of which shall have the same behavior as the original declaration.

2

The function signature `strpbrk(const char*, const char*)` shall be replaced by the two declarations:

```
const char* strpbrk(const char* s1, const char* s2);
      char* strpbrk(      char* s1, const char* s2);
```

both of which shall have the same behavior as the original declaration.

1 The function signature `strrchr(const char*, int)` shall be replaced by the two declarations:

```
const char* strrchr(const char* s, int c);
      char* strrchr(      char* s, int c);
```

both of which shall have the same behavior as the original declaration.

2 The function signature `strstr(const char*, const char*)` shall be replaced by the two declarations:

```
const char* strstr(const char* s1, const char* s2);
      char* strstr(      char* s1, const char* s2);
```

both of which shall have the same behavior as the original declaration.

3

The function signature `memchr(const void*, int, size_t)` shall be replaced by the two declarations:

```
const void* memchr(const void* s, int c, size_t n);
      void* memchr(      void* s, int c, size_t n);
```

both of which shall have the same behavior as the original declaration.

1 The function signature `wcschr(const wchar_t*, wchar_t)` shall be replaced by the two declarations:

```
const wchar_t* wcschr(const wchar_t* s, wchar_t c);
      wchar_t* wcschr(      wchar_t* s, wchar_t c);
```

both of which shall have the same behavior as the original declaration.

2 The function signature `wcspbrk(const wchar_t*, const wchar_t*)` shall be replaced by the two declarations:

```
const wchar_t* wcspbrk(const wchar_t* s1, const wchar_t* s2);
      wchar_t* wcspbrk(      wchar_t* s1, const wchar_t* s2);
```

both of which shall have the same behavior as the original declaration.

3 The function signature `wcsrchr(const wchar_t*, wchar_t)` shall be replaced by the two declarations:

```
const wchar_t* wcsrchr(const wchar_t* s, wchar_t c);
      wchar_t* wcsrchr(      wchar_t* s, wchar_t c);
```

both of which shall have the same behavior as the original declaration.

4 The function signature `wcsstr(const wchar_t*, const wchar_t*)` shall be replaced by the two declarations:

```
const wchar_t* wcsstr(const wchar_t* s1, const wchar_t* s2);
      wchar_t* wcsstr(      wchar_t* s1, const wchar_t* s2);
```

both of which shall have the same behavior as the original declaration.

5 The function signature `wmemchr(const wwchar_t*, int, size_t)` shall be replaced by the two declarations:

```
const wchar_t* wmemchr(const wchar_t* s, wchar_t c, size_t n);
      wchar_t* wmemchr(      wchar_t* s, wchar_t c, size_t n);
```

both of which shall have the same behavior as the original declaration.

6 The functions `strerror` and `strtok` are not required to avoid data races (17.607.0.1).

7

Calling the functions listed in Table 73 with an `mbstate_t*` argument of `NULL` may introduce a data race (17.607.0.1) with other calls to these functions with an `mbstate_t*` argument of `NULL`.

Table 73 — Potential `mbstate_t` data races

mbrlen	mbrtowc	mbsrtowc	mbtowc	wcrtomb
wcsrtomb	wctomb			

See also: ISO C 7.3, 7.10.7, 7.10.8, and 7.11. Amendment 1 4.4, 4.5, and 4.6.

Table 74 — Header `<cctype>` synopsis

Type	Name(s)			
Functions:				
isalnum	isblank	isdigit	isprint	isupper
tolower	isalpha	isgraph	ispunct	isxdigit
toupper	iscntrl	islower	isspace	

Table 75 — Header `<cwctype>` synopsis

Type	Name(s)			
Macro:	WEOF			
Types:	wctrans_t	wctype_t	wint_t	
Functions:				
iswalnum	iswctype	iswprint	iswxdigit	wctrans
iswalpha	iswdigit	iswpunct	towctrans	wctype
iswblank	iswgraph	iswspace	towlower	
iswcntrl	iswlower	iswupper	towupper	

Table 76 — Header `<cstring>` synopsis

Type	Name(s)			
Macro:	NULL <cstring>			
Type:	size_t <cstring>			
Functions:				
memchr	strcat	strcspn	strncpy	strtok
memcmp	strchr	strerror	strpbrk	strxfrm
memcpy	strcmp	strlen	strrchr	
memmove	strcoll	strncat	strspn	
memset	strcpy	strncmp	strstr	

Table 77 — Header `<cwchar>` synopsis

Type	Name(s)			
Macros:	NULL	WCHAR_MAX	WCHAR_MIN	WEOF
Types:	mbstate_t	wint_t	size_t	tm
Functions:				
btowc	mbsinit	vwscanf	wcsncpy	wcstoull
fgetwc	mbsrtowcs	wcrtomb	wcspbrk	wcstoul
fgetws	putwchar	wcscat	wcsrchr	wcsxfrm
fputwc	putwc	wcschr	wcsrtombs	wctob
fputws	swprintf	wcscmp	wcsspn	wmemchr
fwide	swscanf	wcscoll	wcsstr	wmemcmp
fwprintf	ungetwc	wcscpy	wcstod	wmemcpy
fwscanf	vfwprintf	wcscspn	wcstof	wmemmove
getwchar	vfwscanf	wcsftime	wcstok	wmemset
getwc	vswprintf	wcslen	wcstold	wprintf
mbrlen	vswscanf	wcsncat	wcstoll	wscanf
mbrtowc	vwprintf	wcsncmp	wcstol	

Table 78 — Header `<cstdlib>` synopsis

Type	Name(s)		
Macros:	MB_CUR_MAX		
Functions:			
atof	mblen	strtof	strtoul
atoi	mbtowc	strtol	strtoull
atol	mbstowcs	strtold	wctomb
atoll	strtod	strtoll	wcstombs

Table 79 — Header `<cuchar>` synopsis

Type	Name(s)	
Macros:	`__STDC_UTF_16__`	
	`__STDC_UTF_32__`	
Functions:	`mbrtoc16`	`c16rtomb`
	`mbrtoc32`	`c32rtomb`

22 Localization library [localization]

22.1 General [localization.general]

1 This Clause describes components that C++ programs may use to encapsulate (and therefore be more portable when confronting) cultural differences. The locale facility includes internationalization support for character classification and string collation, numeric, monetary, and date/time formatting and parsing, and message retrieval.

2 The following subclauses describe components for locales themselves, the standard facets, and facilities from the ISO C library, as summarized in Table 80.

Table 80 — Localization library summary

Subclause		Header(s)
22.3	Locales	`<locale>`
22.31	Standard `locale` Categories	
22.108	Standard code conversion facets	`<codecvt>`
22.118	C library locales	`<clocale>`

22.2 Header `<locale>` synopsis [locale.syn]

```
namespace std {
  // 22.3.1, locale:
  class locale;
  template <class Facet> const Facet& use_facet(const locale&);
  template <class Facet> bool         has_facet(const locale&) noexcept;

  // 22.25.2, convenience interfaces:
  template <class charT> bool isspace (charT c, const locale& loc);
  template <class charT> bool isprint (charT c, const locale& loc);
  template <class charT> bool iscntrl (charT c, const locale& loc);
  template <class charT> bool isupper (charT c, const locale& loc);
  template <class charT> bool islower (charT c, const locale& loc);
  template <class charT> bool isalpha (charT c, const locale& loc);
  template <class charT> bool isdigit (charT c, const locale& loc);
  template <class charT> bool ispunct (charT c, const locale& loc);
  template <class charT> bool isxdigit(charT c, const locale& loc);
  template <class charT> bool isalnum (charT c, const locale& loc);
  template <class charT> bool isgraph (charT c, const locale& loc);
  template <class charT> bool isblank (charT c, const locale& loc);
  template <class charT> charT toupper(charT c, const locale& loc);
  template <class charT> charT tolower(charT c, const locale& loc);
  template <class Codecvt, class Elem = wchar_t,
    class Wide_alloc = std::allocator<Elem>,
    class Byte_alloc = std::allocator<char> > class wstring_convert;
  template <class Codecvt, class Elem = wchar_t,
     class Tr = char_traits<Elem>> class wbuffer_convert;

  // 22.31.1, ctype:
  class ctype_base;
```

```
  template <class charT> class ctype;
  template <>            class ctype<char>;              // specialization
  template <class charT> class ctype_byname;
  class codecvt_base;
  template <class internT, class externT, class stateT> class codecvt;
  template <class internT, class externT, class stateT> class codecvt_byname;

  // 22.44.1, numeric:
  template <class charT, class InputIterator = istreambuf_iterator<charT> >  class num_get;
  template <class charT, class OutputIterator = ostreambuf_iterator<charT> > class num_put;
  template <class charT> class numpunct;
  template <class charT> class numpunct_byname;

  // 22.102.2, collation:
  template <class charT> class collate;
  template <class charT> class collate_byname;

  // 22.102.3, date and time:
  class time_base;
  template <class charT, class InputIterator = istreambuf_iterator<charT> >
    class time_get;
  template <class charT, class InputIterator = istreambuf_iterator<charT> >
    class time_get_byname;
  template <class charT, class OutputIterator = ostreambuf_iterator<charT> >
    class time_put;
  template <class charT, class OutputIterator = ostreambuf_iterator<charT> >
    class time_put_byname;

  // 22.105.1, money:
  class money_base;
  template <class charT, class InputIterator = istreambuf_iterator<charT> >  class money_get;
  template <class charT, class OutputIterator = ostreambuf_iterator<charT> > class money_put;
  template <class charT, bool Intl = false> class moneypunct;
  template <class charT, bool Intl = false> class moneypunct_byname;

  // 22.107.1, message retrieval:
  class messages_base;
  template <class charT> class messages;
  template <class charT> class messages_byname;
}
```

1 The header `<locale>` defines classes and declares functions that encapsulate and manipulate the information peculiar to a locale.[236]

22.3 Locales [locales]

22.3.1 Class `locale` [locale]

```
namespace std {
  class locale {
  public:
    // types:
    class facet;
    class id;
    typedef int category;
```

236) In this subclause, the type name `struct tm` is an incomplete type that is defined in `<ctime>`.

```
    static const category   // values assigned here are for exposition only
      none     = 0,
      collate  = 0x010, ctype    = 0x020,
      monetary = 0x040, numeric  = 0x080,
      time     = 0x100, messages = 0x200,
      all = collate | ctype | monetary | numeric | time  | messages;

    // construct/copy/destroy:
    locale() noexcept;
    locale(const locale& other) noexcept;
    explicit locale(const char* std_name);
    explicit locale(const string& std_name);
    locale(const locale& other, const char* std_name, category);
    locale(const locale& other, const string& std_name, category);
    template <class Facet> locale(const locale& other, Facet* f);
    locale(const locale& other, const locale& one, category);
    ~locale();                  // not virtual
    const locale& operator=(const locale& other) noexcept;
    template <class Facet> locale combine(const locale& other) const;

    // locale operations:
    basic_string<char>                 name() const;

    bool operator==(const locale& other) const;
    bool operator!=(const locale& other) const;

    template <class charT, class traits, class Allocator>
      bool operator()(const basic_string<charT,traits,Allocator>& s1,
                      const basic_string<charT,traits,Allocator>& s2) const;

    // global locale objects:
    static       locale  global(const locale&);
    static const locale& classic();
  };
}
```

1 Class `locale` implements a type-safe polymorphic set of facets, indexed by facet *type*. In other words, a facet has a dual role: in one sense, it's just a class interface; at the same time, it's an index into a locale's set of facets.

2 Access to the facets of a `locale` is via two function templates, `use_facet<>` and `has_facet<>`.

3

[*Example:* An iostream `operator<<` might be implemented as:[237]

```
template <class charT, class traits>
basic_ostream<charT,traits>&
operator<< (basic_ostream<charT,traits>& s, Date d) {
  typename basic_ostream<charT,traits>::sentry cerberos(s);
  if (cerberos) {
    ios_base::iostate err = ios_base::iostate::goodbit;
    tm tmbuf; d.extract(tmbuf);
    use_facet< time_put<charT,ostreambuf_iterator<charT,traits> > >(
      s.getloc()).put(s, s, s.fill(), err, &tmbuf, 'x');
```

237) Note that in the call to `put` the stream is implicitly converted to an `ostreambuf_iterator<charT,traits>`.

```
    s.setstate(err);            // might throw
  }
  return s;
}
```

— *end example*]

1 In the call to `use_facet<Facet>(loc)`, the type argument chooses a facet, making available all members of the named type. If `Facet` is not present in a locale, it throws the standard exception `bad_cast`. A C++ program can check if a locale implements a particular facet with the function template `has_facet<Facet>()`. User-defined facets may be installed in a locale, and used identically as may standard facets (22.107.2).

2

[*Note:* All locale semantics are accessed via `use_facet<>` and `has_facet<>`, except that:

— A member operator template `operator()(const basic_string<C, T, A>&, const basic_string<C, T, A>&)` is provided so that a locale may be used as a predicate argument to the standard collections, to collate strings.

— Convenient global interfaces are provided for traditional `ctype` functions such as `isdigit()` and `isspace()`, so that given a locale object `loc` a C++ program can call `isspace(c,loc)`. (This eases upgrading existing extractors (27.128.0.1).) — *end note*]

1

Once a facet reference is obtained from a locale object by calling `use_facet<>`, that reference remains usable, and the results from member functions of it may be cached and re-used, as long as some locale object refers to that facet.

1 In successive calls to a locale facet member function on a facet object installed in the same locale, the returned result shall be identical.

2 A `locale` constructed from a name string (such as `"POSIX"`), or from parts of two named locales, has a name; all others do not. Named locales may be compared for equality; an unnamed locale is equal only to (copies of) itself. For an unnamed locale, `locale::name()` returns the string `"*"`.

3 Whether there is one global locale object for the entire program or one global locale object per thread is implementation-defined. Implementations should provide one global locale object per thread. If there is a single global locale object for the entire program, implementations are not required to avoid data races on it (17.607.0.1).

22.6.0.1 `locale` types [locale.types]

22.6.0.1.1 Type `locale::category` [locale.category]

```
typedef int category;
```

1 *Valid* `category` values include the `locale` member bitmask elements `collate`, `ctype`, `monetary`, `numeric`, `time`, and `messages`, each of which represents a single locale category. In addition, `locale` member bitmask constant `none` is defined as zero and represents no category. And `locale` member bitmask constant `all` is defined such that the expression

```
(collate | ctype | monetary | numeric | time | messages | all) == all
```

is `true`, and represents the union of all categories. Further, the expression `(X | Y)`, where `X` and `Y` each represent a single category, represents the union of the two categories.

Table 81 — Locale category facets

Category	Includes facets
collate	`collate<char>`, `collate<wchar_t>`

ctype	`ctype<char>`, `ctype<wchar_t>`
	`codecvt<char,char,mbstate_t>`
	`codecvt<char16_t,char,mbstate_t>`
	`codecvt<char32_t,char,mbstate_t>`
	`codecvt<wchar_t,char,mbstate_t>`

monetary	`moneypunct<char>`, `moneypunct<wchar_t>`
	`moneypunct<char,true>`, `moneypunct<wchar_t,true>`
	`money_get<char>`, `money_get<wchar_t>`
	`money_put<char>`, `money_put<wchar_t>`

numeric	`numpunct<char>`, `numpunct<wchar_t>`
	`num_get<char>`, `num_get<wchar_t>`
	`num_put<char>`, `num_put<wchar_t>`

time	`time_get<char>`, `time_get<wchar_t>` `time_put<char>`, `time_put<wchar_t>`
messages	`messages<char>`, `messages<wchar_t>`

2 `locale` member functions expecting a `category` argument require one of the `category` values defined above, or the union of two or more such values. Such a `category` value identifies a set of locale categories. Each locale category, in turn, identifies a set of locale facets, including at least those shown in Table 81.

3 For any locale `loc` either constructed, or returned by `locale::classic()`, and any facet `Facet` shown in Table 81, `has_facet<Facet>(loc)` is true. Each `locale` member function which takes a `locale::category` argument operates on the corresponding set of facets.

4 An implementation is required to provide those specializations for facet templates identified as members of a category, and for those shown in Table 82.

5 The provided implementation of members of facets `num_get<charT>` and `num_put<charT>` calls `use_facet<F>(l)` only for facet `F` of types `numpunct<charT>` and `ctype<charT>`, and for locale `l` the value obtained by calling member `getloc()` on the `ios_base&` argument to these functions.

6 In declarations of facets, a template parameter with name `InputIterator` or `OutputIterator` indicates the set of all possible specializations on parameters that satisfy the requirements of an Input Iterator or an Output Iterator, respectively (24.2). A template parameter with name `C` represents the set of types containing `char`, `wchar_t`, and any other implementation-defined character types that satisfy the requirements for a character on which any of the iostream components can be instantiated. A template parameter with name `International` represents the set of all possible specializations on a bool parameter.

22.6.0.1.2 Class `locale::facet` [locale.facet]

```
namespace std {
  class locale::facet {
  protected:
    explicit facet(size_t refs = 0);
    virtual ~facet();
    facet(const facet&) = delete;
    void operator=(const facet&) = delete;
  };
}
```

1 Template parameters in this Clause which are required to be facets are those named `Facet` in declarations. A

Table 82 — Required specializations

Category	Includes facets
collate	`collate_byname<char>`, `collate_byname<wchar_t>`

ctype	`ctype_byname<char>`, `ctype_byname<wchar_t>`
	`codecvt_byname<char,char,mbstate_t>`
	`codecvt_byname<char16_t,char,mbstate_t>`
	`codecvt_byname<char32_t,char,mbstate_t>`
	`codecvt_byname<wchar_t,char,mbstate_t>`

monetary	`moneypunct_byname<char,International>`
	`moneypunct_byname<wchar_t,International>`
	`money_get<C,InputIterator>`
	`money_put<C,OutputIterator>`

numeric	`numpunct_byname<char>`, `numpunct_byname<wchar_t>` `num_get<C,InputIterator>`, `num_put<C,OutputIterator>`

time	`time_get<char,InputIterator>`
	`time_get_byname<char,InputIterator>`
	`time_get<wchar_t,InputIterator>`
	`time_get_byname<wchar_t,InputIterator>`
	`time_put<char,OutputIterator>`
	`time_put_byname<char,OutputIterator>`
	`time_put<wchar_t,OutputIterator>`
	`time_put_byname<wchar_t,OutputIterator>`

messages	`messages_byname<char>`, `messages_byname<wchar_t>`

program that passes a type that is *not* a facet, or a type that refers to a volatile-qualified facet, as an (explicit or deduced) template parameter to a locale function expecting a facet, is ill-formed. A const-qualified facet is a valid template argument to any locale function that expects a Facet template parameter.

2

The `refs` argument to the constructor is used for lifetime management.

— For `refs == 0`, the implementation performs `delete static_cast<locale::facet*>(f)` (where `f` is a pointer to the facet) when the last `locale` object containing the facet is destroyed; for `refs == 1`, the implementation never destroys the facet.

1 Constructors of all facets defined in this Clause take such an argument and pass it along to their `facet` base class constructor. All one-argument constructors defined in this Clause are *explicit*, preventing their participation in automatic conversions.

2

For some standard facets a standard "…`_byname`" class, derived from it, implements the virtual function semantics equivalent to that facet of the locale constructed by `locale(const char*)` with the same name. Each such facet provides a constructor that takes a `const char*` argument, which names the locale, and a `refs` argument, which is passed to the base class constructor. Each such facet also provides a constructor that takes a `string` argument `str` and a `refs` argument, which has the same effect as calling the first constructor with the two arguments `str.c_str()` and `refs`. If there is no "…`_byname`" version of a facet, the base class implements named locale semantics itself by reference to other facets.

22.14.0.0.1 Class `locale::id` [locale.id]

```
namespace std {
  class locale::id {
```

```
  public:
    id();
    void operator=(const id&) = delete;
    id(const id&) = delete;
  };
}
```

1 The class locale::id provides identification of a locale facet interface, used as an index for lookup and to encapsulate initialization.

2 [*Note:* Because facets are used by iostreams, potentially while static constructors are running, their initialization cannot depend on programmed static initialization. One initialization strategy is for `locale` to initialize each facet's `id` member the first time an instance of the facet is installed into a locale. This depends only on static storage being zero before constructors run (3.1937.1). — *end note*]

22.14.0.1 `locale` constructors and destructor [locale.cons]

`locale() noexcept;`

1

Default constructor: a snapshot of the current global locale.

1

Effects: Constructs a copy of the argument last passed to `locale::global(locale&)`, if it has been called; else, the resulting facets have virtual function semantics identical to those of `locale::classic()`. [*Note:* This constructor is commonly used as the default value for arguments of functions that take a `const locale&` argument. — *end note*]

`locale(const locale& other) noexcept;`

1 *Effects:* Constructs a locale which is a copy of `other`.

`explicit locale(const char* std_name);`

2

Effects: Constructs a locale using standard C locale names, e.g., `"POSIX"`. The resulting locale implements semantics defined to be associated with that name.

1

Throws: `runtime_error` if the argument is not valid, or is null.

1

Remarks: The set of valid string argument values is `"C"`, `""`, and any implementation-defined values.

`explicit locale(const string& std_name);`

1 *Effects:* The same as `locale(std_name.c_str())`.

`locale(const locale& other, const char* std_name, category);`

2 *Effects:* Constructs a locale as a copy of `other` except for the facets identified by the `category` argument, which instead implement the same semantics as `locale(std_name)`.

3 *Throws:* `runtime_error` if the argument is not valid, or is null.

4 *Remarks:* The locale has a name if and only if `other` has a name.

`locale(const locale& other, const string& std_name, category cat);`

5 *Effects:* The same as `locale(other, std_name.c_str(), cat)`.

```
template <class Facet> locale(const locale& other, Facet* f);
```

6 *Effects:* Constructs a locale incorporating all facets from the first argument except that of type `Facet`, and installs the second argument as the remaining facet. If `f` is null, the resulting object is a copy of `other`.

7 *Remarks:* The resulting locale has no name.

```
locale(const locale& other, const locale& one, category cats);
```

8 *Effects:* Constructs a locale incorporating all facets from the first argument except those that implement `cats`, which are instead incorporated from the second argument.

9 *Remarks:* The resulting locale has a name if and only if the first two arguments have names.

```
const locale& operator=(const locale& other) noexcept;
```

10 *Effects:* Creates a copy of `other`, replacing the current value.

11 *Returns:* `*this`

```
~locale();
```

12 A non-virtual destructor that throws no exceptions.

22.20.0.1 `locale` members [locale.members]

```
template <class Facet> locale combine(const locale& other) const;
```

1 *Effects:* Constructs a locale incorporating all facets from `*this` except for that one facet of `other` that is identified by `Facet`.

2 *Returns:* The newly created locale.

3 *Throws:* `runtime_error` if `has_facet<Facet>(other)` is false.

4 *Remarks:* The resulting locale has no name.

```
basic_string<char> name() const;
```

5 *Returns:* The name of `*this`, if it has one; otherwise, the string `"*"`. If `*this` has a name, then `locale(name().c_str())` is equivalent to `*this`. Details of the contents of the resulting string are otherwise implementation-defined.

22.20.0.2 `locale` operators [locale.operators]

```
bool operator==(const locale& other) const;
```

1 *Returns:* `true` if both arguments are the same locale, or one is a copy of the other, or each has a name and the names are identical; `false` otherwise.

```
bool operator!=(const locale& other) const;
```

2 *Returns:* The result of the expression: `!(*this == other)`.

```
template <class charT, class traits, class Allocator>
  bool operator()(const basic_string<charT,traits,Allocator>& s1,
                  const basic_string<charT,traits,Allocator>& s2) const;
```

3 *Effects:* Compares two strings according to the `collate<charT>` facet.

4 *Remarks:* This member operator template (and therefore `locale` itself) satisfies requirements for a comparator predicate template argument (Clause 25) applied to strings.

5 *Returns:* The result of the following expression:

```
use_facet< collate<charT> >(*this).compare
  (s1.data(), s1.data()+s1.size(), s2.data(), s2.data()+s2.size()) < 0;
```

6 [*Example:* A vector of strings `v` can be collated according to collation rules in locale `loc` simply by (25.144.1, 23.696.1):

```
std::sort(v.begin(), v.end(), loc);
```

— *end example*]

22.20.0.3 `locale` static members [locale.statics]

`static locale global(const locale& loc);`

1

Sets the global locale to its argument.

1

Effects: Causes future calls to the constructor `locale()` to return a copy of the argument. If the argument has a name, does

```
std::setlocale(LC_ALL, loc.name().c_str());
```

otherwise, the effect on the C locale, if any, is implementation-defined. No library function other than `locale::global()` shall affect the value returned by `locale()`. [*Note:* See 22.118 for data race considerations when `setlocale` is invoked. — *end note*]

1

Returns: The previous value of `locale()`.

`static const locale& classic();`

1 The `"C"` locale.

2

Returns: A locale that implements the classic `"C"` locale semantics, equivalent to the value `locale("C")`.

1

Remarks: This locale, its facets, and their member functions, do not change with time.

22.25.1 `locale` globals [locale.global.templates]

`template <class Facet> const Facet& use_facet(const locale& loc);`

1 *Requires:* `Facet` is a facet class whose definition contains the public static member `id` as defined in 22.6.0.1.2.

2 *Returns:* A reference to the corresponding facet of `loc`, if present.

3 *Throws:* `bad_cast` if `has_facet<Facet>(loc)` is `false`.

4 *Remarks:* The reference returned remains valid at least as long as any copy of `loc` exists.

`template <class Facet> bool has_facet(const locale& loc) noexcept;`

5 *Returns:* True if the facet requested is present in `loc`; otherwise false.

22.25.2 Convenience interfaces [locale.convenience]

22.25.2.1 Character classification [classification]

```
template <class charT> bool isspace (charT c, const locale& loc);
template <class charT> bool isprint (charT c, const locale& loc);
template <class charT> bool iscntrl (charT c, const locale& loc);
template <class charT> bool isupper (charT c, const locale& loc);
template <class charT> bool islower (charT c, const locale& loc);
template <class charT> bool isalpha (charT c, const locale& loc);
template <class charT> bool isdigit (charT c, const locale& loc);
template <class charT> bool ispunct (charT c, const locale& loc);
template <class charT> bool isxdigit(charT c, const locale& loc);
template <class charT> bool isalnum (charT c, const locale& loc);
template <class charT> bool isgraph (charT c, const locale& loc);
template <class charT> bool isblank (charT c, const locale& loc);
```

1

Each of these functions `is`***F*** returns the result of the expression:

```
use_facet< ctype<charT> >(loc).is(ctype_base::F, c)
```

where ***F*** is the `ctype_base::mask` value corresponding to that function (22.31.1).[238]

22.26.0.1 Conversions [conversions]

22.26.0.1.1 Character conversions [conversions.character]

```
template <class charT> charT toupper(charT c, const locale& loc);
```

1

Returns: `use_facet<ctype<charT> >(loc).toupper(c)`.

```
template <class charT> charT tolower(charT c, const locale& loc);
```

1 *Returns:* `use_facet<ctype<charT> >(loc).tolower(c)`.

22.27.0.0.1 `string` conversions [conversions.string]

1

Class template `wstring_convert` performs conversions between a wide string and a byte string. It lets you specify a code conversion facet (like class template `codecvt`) to perform the conversions, without affecting any streams or locales. [*Example:* If you want to use the code conversion facet `codecvt_utf8` to output to `cout` a UTF-8 multibyte sequence corresponding to a wide string, but you don't want to alter the locale for `cout`, you can write something like:

```
wstring_convert<std::codecvt_utf8<wchar_t>> myconv;
std::string mbstring = myconv.to_bytes(L"Hello\n");
std::cout << mbstring;
```

— *end example*]

1 **Class template `wstring_convert` synopsis**

```
namespace std {
template<class Codecvt, class Elem = wchar_t,
    class Wide_alloc = std::allocator<Elem>,
    class Byte_alloc = std::allocator<char> > class wstring_convert {
```

238) When used in a loop, it is faster to cache the `ctype<>` facet and use it directly, or use the vector form of `ctype<>::is`.

```
  public:
    typedef std::basic_string<char, char_traits<char>, Byte_alloc> byte_string;
    typedef std::basic_string<Elem, char_traits<Elem>, Wide_alloc> wide_string;
    typedef typename Codecvt::state_type state_type;
    typedef typename wide_string::traits_type::int_type int_type;

    explicit wstring_convert(Codecvt* pcvt = new Codecvt);
    wstring_convert(Codecvt* pcvt, state_type state);
    explicit wstring_convert(const byte_string& byte_err,
                             const wide_string& wide_err = wide_string());
    ~wstring_convert();

    wstring_convert(const wstring_convert&) = delete;
    wstring_convert& operator=(const wstring_convert&) = delete;

    wide_string from_bytes(char byte);
    wide_string from_bytes(const char* ptr);
    wide_string from_bytes(const byte_string& str);
    wide_string from_bytes(const char* first, const char* last);

    byte_string to_bytes(Elem wchar);
    byte_string to_bytes(const Elem* wptr);
    byte_string to_bytes(const wide_string& wstr);
    byte_string to_bytes(const Elem* first, const Elem* last);

    size_t converted() const noexcept;
    state_type state() const;
  private:
    byte_string byte_err_string;    // exposition only
    wide_string wide_err_string;    // exposition only
    Codecvt* cvtptr;                // exposition only
    state_type cvtstate;            // exposition only
    size_t cvtcount;                // exposition only
  };
}
```

2 The class template describes an object that controls conversions between wide string objects of class `std::basic_string<Elem, char_traits<Elem>, Wide_alloc>` and byte string objects of class `std::basic_string<char, char_traits<char>, Byte_alloc>`. The class template defines the types `wide_string` and `byte_string` as synonyms for these two types. Conversion between a sequence of `Elem` values (stored in a `wide_string` object) and multibyte sequences (stored in a `byte_string` object) is performed by an object of class `Codecvt`, which meets the requirements of the standard code-conversion facet `std::codecvt<Elem, char, std::mbstate_t>`.

3 An object of this class template stores:

(3.1) — `byte_err_string` — a byte string to display on errors

(3.2) — `wide_err_string` — a wide string to display on errors

(3.3) — `cvtptr` — a pointer to the allocated conversion object (which is freed when the `wstring_convert` object is destroyed)

(3.4) — `cvtstate` — a conversion state object

(3.5) — `cvtcount` — a conversion count

```
typedef std::basic_string<char, char_traits<char>, Byte_alloc> byte_string;
```

4 The type shall be a synonym for `std::basic_string<char, char_traits<char>, Byte_alloc>`

```
size_t converted() const noexcept;
```

5 *Returns:* `cvtcount`.

```
wide_string from_bytes(char byte);
wide_string from_bytes(const char* ptr);
wide_string from_bytes(const byte_string& str);
wide_string from_bytes(const char* first, const char* last);
```

6 *Effects:* The first member function shall convert the single-element sequence `byte` to a wide string. The second member function shall convert the null-terminated sequence beginning at `ptr` to a wide string. The third member function shall convert the sequence stored in `str` to a wide string. The fourth member function shall convert the sequence defined by the range `[first,last)` to a wide string.

7 In all cases:

(7.1) — If the `cvtstate` object was not constructed with an explicit value, it shall be set to its default value (the initial conversion state) before the conversion begins. Otherwise it shall be left unchanged.

(7.2) — The number of input elements successfully converted shall be stored in `cvtcount`.

8

Returns: If no conversion error occurs, the member function shall return the converted wide string. Otherwise, if the object was constructed with a wide-error string, the member function shall return the wide-error string. Otherwise, the member function throws an object of class `std::range_error`.

```
typedef typename wide_string::traits_type::int_type int_type;
```

The type shall be a synonym for `wide_string::traits_type::int_type`.

```
state_type state() const;
```

1 returns `cvtstate`.

```
typedef typename Codecvt::state_type state_type;
```

2 The type shall be a synonym for `Codecvt::state_type`.

```
byte_string to_bytes(Elem wchar);
byte_string to_bytes(const Elem* wptr);
byte_string to_bytes(const wide_string& wstr);
byte_string to_bytes(const Elem* first, const Elem* last);
```

3 *Effects:* The first member function shall convert the single-element sequence `wchar` to a byte string. The second member function shall convert the null-terminated sequence beginning at `wptr` to a byte string. The third member function shall convert the sequence stored in `wstr` to a byte string. The fourth member function shall convert the sequence defined by the range `[first,last)` to a byte string.

4 In all cases:

(4.1) — If the `cvtstate` object was not constructed with an explicit value, it shall be set to its default value (the initial conversion state) before the conversion begins. Otherwise it shall be left unchanged.

(4.2) — The number of input elements successfully converted shall be stored in `cvtcount`.

5 *Returns:* If no conversion error occurs, the member function shall return the converted byte string. Otherwise, if the object was constructed with a byte-error string, the member function shall return the byte-error string. Otherwise, the member function shall throw an object of class `std::range_error`.

```
typedef std::basic_string<Elem, char_traits<Elem>, Wide_alloc> wide_string;
```

6 The type shall be a synonym for `std::basic_string<Elem, char_traits<Elem>, Wide_alloc>`.

```
explicit wstring_convert(Codecvt* pcvt = new Codecvt);
wstring_convert(Codecvt* pcvt, state_type state);
explicit wstring_convert(const byte_string& byte_err,
    const wide_string& wide_err = wide_string());
```

7 *Requires:* For the first and second constructors, `pcvt != nullptr`.

8 *Effects:* The first constructor shall store `pcvt` in `cvtptr` and default values in `cvtstate`, `byte_err_string`, and `wide_err_string`. The second constructor shall store `pcvt` in `cvtptr`, `state` in `cvtstate`, and default values in `byte_err_string` and `wide_err_string`; moreover the stored state shall be retained between calls to `from_bytes` and `to_bytes`. The third constructor shall store `new Codecvt` in `cvtptr`, `state_type()` in `cvtstate`, `byte_err` in `byte_err_string`, and `wide_err` in `wide_err_string`.

```
~wstring_convert();
```

9 *Effects:* The destructor shall delete `cvtptr`.

22.30.0.0.1 Buffer conversions [conversions.buffer]

1 Class template `wbuffer_convert` looks like a wide stream buffer, but performs all its I/O through an underlying byte stream buffer that you specify when you construct it. Like class template `wstring_convert`, it lets you specify a code conversion facet to perform the conversions, without affecting any streams or locales.

2 **Class template `wbuffer_convert` synopsis**

```
namespace std {
template<class Codecvt,
    class Elem = wchar_t,
    class Tr = std::char_traits<Elem> >
  class wbuffer_convert
    : public std::basic_streambuf<Elem, Tr> {
  public:
    typedef typename Codecvt::state_type state_type;

    explicit wbuffer_convert(std::streambuf* bytebuf = 0,
                             Codecvt* pcvt = new Codecvt,
                             state_type state = state_type());

    ~wbuffer_convert();

    wbuffer_convert(const wbuffer_convert&) = delete;
    wbuffer_convert& operator=(const wbuffer_convert&) = delete;

    std::streambuf* rdbuf() const;
    std::streambuf* rdbuf(std::streambuf* bytebuf);

    state_type state() const;

  private:
    std::streambuf* bufptr;          // exposition only
    Codecvt* cvtptr;                 // exposition only
    state_type cvtstate;             // exposition only
  };
}
```

3 The class template describes a stream buffer that controls the transmission of elements of type `Elem`, whose character traits are described by the class `Tr`, to and from a byte stream buffer of type `std::streambuf`. Conversion between a sequence of `Elem` values and multibyte sequences is performed by an object of class `Codecvt`, which shall meet the requirements of the standard code-conversion facet `std::codecvt<Elem, char, std::mbstate_t>`.

4 An object of this class template stores:

(4.1) — `bufptr` — a pointer to its underlying byte stream buffer

(4.2) — `cvtptr` — a pointer to the allocated conversion object (which is freed when the `wbuffer_convert` object is destroyed)

(4.3) — `cvtstate` — a conversion state object

```
state_type state() const;
```

5 *Returns:* `cvtstate`.

```
std::streambuf* rdbuf() const;
```

6 *Returns:* `bufptr`.

```
std::streambuf* rdbuf(std::streambuf* bytebuf);
```

7 *Effects:* stores `bytebuf` in `bufptr`.

8 *Returns:* The previous value of `bufptr`.

```
typedef typename Codecvt::state_type state_type;
```

9 The type shall be a synonym for `Codecvt::state_type`.

```
explicit wbuffer_convert(std::streambuf* bytebuf = 0,
    Codecvt* pcvt = new Codecvt, state_type state = state_type());
```

10 *Requires:* `pcvt != nullptr`.

11 *Effects:* The constructor constructs a stream buffer object, initializes `bufptr` to `bytebuf`, initializes `cvtptr` to `pcvt`, and initializes `cvtstate` to `state`.

```
~wbuffer_convert();
```

12 *Effects:* The destructor shall delete `cvtptr`.

22.31 Standard `locale` categories [locale.categories]

1 Each of the standard categories includes a family of facets. Some of these implement formatting or parsing of a datum, for use by standard or users' iostream operators `<<` and `>>`, as members `put()` and `get()`, respectively. Each such member function takes an `ios_base&` argument whose members `flags()`, `precision()`, and `width()`, specify the format of the corresponding datum (27.58.1). Those functions which need to use other facets call its member `getloc()` to retrieve the locale imbued there. Formatting facets use the character argument `fill` to fill out the specified width where necessary.

2 The `put()` members make no provision for error reporting. (Any failures of the OutputIterator argument must be extracted from the returned iterator.) The `get()` members take an `ios_base::iostate&` argument whose value they ignore, but set to `ios_base::failbit` in case of a parse error.

3 Within this clause it is unspecified whether one virtual function calls another virtual function.

22.31.1 The ctype category [category.ctype]

```
namespace std {
  class ctype_base {
  public:
    typedef T mask;

    // numeric values are for exposition only.
    static const mask space = 1 << 0;
    static const mask print = 1 << 1;
    static const mask cntrl = 1 << 2;
    static const mask upper = 1 << 3;
    static const mask lower = 1 << 4;
    static const mask alpha = 1 << 5;
    static const mask digit = 1 << 6;
    static const mask punct = 1 << 7;
    static const mask xdigit = 1 << 8;
    static const mask blank = 1 << 9;
    static const mask alnum = alpha | digit;
    static const mask graph = alnum | punct;
  };
}
```

1 The type mask is a bitmask type (17.20.0.0.1).

22.31.1.1 Class template ctype [locale.ctype]

```
namespace std {
  template <class charT>
  class ctype : public locale::facet, public ctype_base {
  public:
    typedef charT char_type;

    explicit ctype(size_t refs = 0);

    bool         is(mask m, charT c) const;
    const charT* is(const charT* low, const charT* high, mask* vec) const;
    const charT* scan_is(mask m,
                         const charT* low, const charT* high) const;
    const charT* scan_not(mask m,
                          const charT* low, const charT* high) const;
    charT        toupper(charT c) const;
    const charT* toupper(charT* low, const charT* high) const;
    charT        tolower(charT c) const;
    const charT* tolower(charT* low, const charT* high) const;

    charT        widen(char c) const;
    const char*  widen(const char* low, const char* high, charT* to) const;
    char         narrow(charT c, char dfault) const;
    const charT* narrow(const charT* low, const charT*, char dfault,
                        char* to) const;

    static locale::id id;

  protected:
   ~ctype();
    virtual bool         do_is(mask m, charT c) const;
```

```
    virtual const charT* do_is(const charT* low, const charT* high,
                               mask* vec) const;
    virtual const charT* do_scan_is(mask m,
                                    const charT* low, const charT* high) const;
    virtual const charT* do_scan_not(mask m,
                                     const charT* low, const charT* high) const;
    virtual charT        do_toupper(charT) const;
    virtual const charT* do_toupper(charT* low, const charT* high) const;
    virtual charT        do_tolower(charT) const;
    virtual const charT* do_tolower(charT* low, const charT* high) const;
    virtual charT        do_widen(char) const;
    virtual const char*  do_widen(const char* low, const char* high,
                                  charT* dest) const;
    virtual char         do_narrow(charT, char dfault) const;
    virtual const charT* do_narrow(const charT* low, const charT* high,
                                   char dfault, char* dest) const;
  };
}
```

1 Class `ctype` encapsulates the C library `<cctype>` features. `istream` members are required to use `ctype<>` for character classing during input parsing.

2 The specializations required in Table 81 (22.6.0.1.1), namely `ctype<char>` and `ctype<wchar_t>`, implement character classing appropriate to the implementation's native character set.

22.31.1.1.1 `ctype` members [locale.ctype.members]

```
bool         is(mask m, charT c) const;
const charT* is(const charT* low, const charT* high,
                mask* vec) const;
```

1 *Returns:* `do_is(m,c)` or `do_is(low,high,vec)`

```
const charT* scan_is(mask m,
                     const charT* low, const charT* high) const;
```

2 *Returns:* `do_scan_is(m,low,high)`

```
const charT* scan_not(mask m,
                      const charT* low, const charT* high) const;
```

3 *Returns:* `do_scan_not(m,low,high)`

```
charT        toupper(charT) const;
const charT* toupper(charT* low, const charT* high) const;
```

4 *Returns:* `do_toupper(c)` or `do_toupper(low,high)`

```
charT        tolower(charT c) const;
const charT* tolower(charT* low, const charT* high) const;
```

5 *Returns:* `do_tolower(c)` or `do_tolower(low,high)`

```
charT       widen(char c) const;
const char* widen(const char* low, const char* high, charT* to) const;
```

6 *Returns:* `do_widen(c)` or `do_widen(low,high,to)`

```
char         narrow(charT c, char dfault) const;
const charT* narrow(const charT* low, const charT*, char dfault,
                    char* to) const;
```

7 *Returns:* `do_narrow(c,dfault)` or `do_narrow(low,high,dfault,to)`

22.31.1.1.2 `ctype` virtual functions [locale.ctype.virtuals]

```
bool         do_is(mask m, charT c) const;
const charT* do_is(const charT* low, const charT* high,
                   mask* vec) const;
```

1 *Effects:* Classifies a character or sequence of characters. For each argument character, identifies a value `M` of type `ctype_base::mask`. The second form identifies a value `M` of type `ctype_base::mask` for each `*p` where `(low<=p && p<high)`, and places it into `vec[p-low]`.

2 *Returns:* The first form returns the result of the expression `(M & m) != 0`; i.e., `true` if the character has the characteristics specified. The second form returns `high`.

```
const charT* do_scan_is(mask m,
                        const charT* low, const charT* high) const;
```

3 *Effects:* Locates a character in a buffer that conforms to a classification `m`.

4 *Returns:* The smallest pointer `p` in the range `[low, high)` such that `is(m,*p)` would return `true`; otherwise, returns `high`.

```
const charT* do_scan_not(mask m,
                         const charT* low, const charT* high) const;
```

5 *Effects:* Locates a character in a buffer that fails to conform to a classification `m`.

6 *Returns:* The smallest pointer `p`, if any, in the range `[low,high)` such that `is(m,*p)` would return `false`; otherwise, returns `high`.

```
charT        do_toupper(charT c) const;
const charT* do_toupper(charT* low, const charT* high) const;
```

7 *Effects:* Converts a character or characters to upper case. The second form replaces each character `*p` in the range `[low,high)` for which a corresponding upper-case character exists, with that character.

8 *Returns:* The first form returns the corresponding upper-case character if it is known to exist, or its argument if not. The second form returns `high`.

```
charT        do_tolower(charT c) const;
const charT* do_tolower(charT* low, const charT* high) const;
```

9 *Effects:* Converts a character or characters to lower case. The second form replaces each character `*p` in the range `[low,high)` and for which a corresponding lower-case character exists, with that character.

10 *Returns:* The first form returns the corresponding lower-case character if it is known to exist, or its argument if not. The second form returns `high`.

```
charT        do_widen(char c) const;
const char*  do_widen(const char* low, const char* high,
                      charT* dest) const;
```

11 *Effects:* Applies the simplest reasonable transformation from a `char` value or sequence of `char` values to the corresponding `charT` value or values.[239] The only characters for which unique transformations are required are those in the basic source character set (2.154).

For any named `ctype` category with a `ctype<charT>` facet `ctc` and valid `ctype_base::mask` value `M`, `(ctc.is(M, c) || !is(M, do_widen(c)) )` is `true`.[240]

The second form transforms each character `*p` in the range `[low,high)`, placing the result in `dest[p-low]`.

12 *Returns:* The first form returns the transformed value. The second form returns `high`.

```
char         do_narrow(charT c, char dfault) const;
const charT* do_narrow(const charT* low, const charT* high,
                       char dfault, char* dest) const;
```

13 *Effects:* Applies the simplest reasonable transformation from a `charT` value or sequence of `charT` values to the corresponding `char` value or values.

For any character `c` in the basic source character set (2.154) the transformation is such that

```
do_widen(do_narrow(c,0)) == c
```

For any named `ctype` category with a `ctype<char>` facet `ctc` however, and `ctype_base::mask` value `M`,

```
(is(M,c) || !ctc.is(M, do_narrow(c,dfault)) )
```

is `true` (unless `do_narrow` returns `dfault`). In addition, for any digit character `c`, the expression `(do_narrow(c, dfault) - '0')` evaluates to the digit value of the character. The second form transforms each character `*p` in the range `[low,high)`, placing the result (or `dfault` if no simple transformation is readily available) in `dest[p-low]`.

14 *Returns:* The first form returns the transformed value; or `dfault` if no mapping is readily available. The second form returns `high`.

22.31.1.2 Class template `ctype_byname` [locale.ctype.byname]

```
namespace std {
  template <class charT>
  class ctype_byname : public ctype<charT> {
  public:
    typedef typename ctype<charT>::mask mask;
    explicit ctype_byname(const char*, size_t refs = 0);
    explicit ctype_byname(const string&, size_t refs = 0);
  protected:
   ~ctype_byname();
  };
}
```

22.31.1.3 `ctype` specializations [facet.ctype.special]

```
namespace std {
  template <> class ctype<char>
    : public locale::facet, public ctype_base {
  public:
    typedef char char_type;
```

239) The char argument of `do_widen` is intended to accept values derived from character literals for conversion to the locale's encoding.

240) In other words, the transformed character is not a member of any character classification that `c` is not also a member of.

```
    explicit ctype(const mask* tab = 0, bool del = false,
                   size_t refs = 0);

    bool is(mask m, char c) const;
    const char* is(const char* low, const char* high, mask* vec) const;
    const char* scan_is (mask m,
                         const char* low, const char* high) const;
    const char* scan_not(mask m,
                         const char* low, const char* high) const;

    char        toupper(char c) const;
    const char* toupper(char* low, const char* high) const;
    char        tolower(char c) const;
    const char* tolower(char* low, const char* high) const;

    char  widen(char c) const;
    const char* widen(const char* low, const char* high, char* to) const;
    char  narrow(char c, char dfault) const;
    const char* narrow(const char* low, const char* high, char dfault,
                       char* to) const;

    static locale::id id;
    static const size_t table_size = implementation-defined;

    const mask* table() const noexcept;
    static const mask* classic_table() noexcept;

  protected:
   ~ctype();
    virtual char        do_toupper(char c) const;
    virtual const char* do_toupper(char* low, const char* high) const;
    virtual char        do_tolower(char c) const;
    virtual const char* do_tolower(char* low, const char* high) const;

    virtual char        do_widen(char c) const;
    virtual const char* do_widen(const char* low,
                                 const char* high,
                                 char* to) const;
    virtual char        do_narrow(char c, char dfault) const;
    virtual const char* do_narrow(const char* low,
                                  const char* high,
                                  char dfault, char* to) const;
  };
}
```

1 A specialization `ctype<char>` is provided so that the member functions on type `char` can be implemented `inline`.[241] The implementation-defined value of member `table_size` is at least 256.

22.31.1.3.1 `ctype<char>` destructor [facet.ctype.char.dtor]

```
~ctype();
```

241) Only the `char` (not `unsigned char` and `signed char`) form is provided. The specialization is specified in the standard, and not left as an implementation detail, because it affects the derivation interface for `ctype<char>`.

1 *Effects:* If the constructor's first argument was nonzero, and its second argument was true, does `delete [] table()`.

22.31.1.3.2 `ctype<char>` members [facet.ctype.char.members]

1 In the following member descriptions, for `unsigned char` values `v` where `v >= table_size`, `table()[v]` is assumed to have an implementation-specific value (possibly different for each such value `v`) without performing the array lookup.

```
explicit ctype(const mask* tbl = 0, bool del = false,
               size_t refs = 0);
```

2 *Requires:* `tbl` either 0 or an array of at least `table_size` elements.

3 *Effects:* Passes its `refs` argument to its base class constructor.

```
bool        is(mask m, char c) const;
const char* is(const char* low, const char* high,
               mask* vec) const;
```

4 *Effects:* The second form, for all `*p` in the range `[low,high)`, assigns into `vec[p-low]` the value `table()[(unsigned char)*p]`.

5 *Returns:* The first form returns `table()[(unsigned char)c] & m`; the second form returns `high`.

```
const char* scan_is(mask m,
                    const char* low, const char* high) const;
```

6 *Returns:* The smallest `p` in the range `[low,high)` such that

```
table()[(unsigned char) *p] & m
```

is `true`.

```
const char* scan_not(mask m,
                     const char* low, const char* high) const;
```

7 *Returns:* The smallest `p` in the range `[low,high)` such that

```
table()[(unsigned char) *p] & m
```

is `false`.

```
char        toupper(char c) const;
const char* toupper(char* low, const char* high) const;
```

8 *Returns:* `do_toupper(c)` or `do_toupper(low,high)`, respectively.

```
char        tolower(char c) const;
const char* tolower(char* low, const char* high) const;
```

9 *Returns:* `do_tolower(c)` or `do_tolower(low,high)`, respectively.

```
char  widen(char c) const;
const char* widen(const char* low, const char* high,
    char* to) const;
```

10 *Returns:* `do_widen(c)` or `do_widen(low, high, to)`, respectively.

```
char        narrow(char c, char dfault) const;
const char* narrow(const char* low, const char* high,
                   char dfault, char* to) const;
```

1 *Returns:* `do_narrow(c, dfault)` or `do_narrow(low, high, dfault, to)`, respectively.

```
const mask* table() const noexcept;
```

2 *Returns:* The first constructor argument, if it was non-zero, otherwise `classic_table()`.

22.32.0.0.1 `ctype<char>` static members [facet.ctype.char.statics]

```
static const mask* classic_table() noexcept;
```

1

Returns: A pointer to the initial element of an array of size `table_size` which represents the classifications of characters in the "C" locale.

22.34.0.0.1 `ctype<char>` virtual functions [facet.ctype.char.virtuals]

```
char        do_toupper(char) const;
const char* do_toupper(char* low, const char* high) const;
char        do_tolower(char) const;
const char* do_tolower(char* low, const char* high) const;

virtual char        do_widen(char c) const;
virtual const char* do_widen(const char* low,
                             const char* high,
                             char* to) const;
virtual char        do_narrow(char c, char dfault) const;
virtual const char* do_narrow(const char* low,
                              const char* high,
                              char dfault, char* to) const;
```

These functions are described identically as those members of the same name in the `ctype` class template (22.31.1.1.1).

22.34.0.1 Class template `codecvt` [locale.codecvt]

```
namespace std {
  class codecvt_base {
  public:
    enum result { ok, partial, error, noconv };
  };

  template <class internT, class externT, class stateT>
  class codecvt : public locale::facet, public codecvt_base {
  public:
    typedef internT  intern_type;
    typedef externT  extern_type;
    typedef stateT state_type;

    explicit codecvt(size_t refs = 0);

    result out(stateT& state,
               const internT* from, const internT* from_end, const internT*& from_next,
               externT*   to,       externT* to_end, externT*& to_next) const;
    result unshift(stateT& state,
```

```
                externT*   to,        externT* to_end, externT*& to_next) const;
  result in(stateT& state,
            const externT* from, const externT* from_end, const externT*& from_next,
            internT*   to,       internT* to_end, internT*& to_next) const;
  int encoding() const noexcept;
  bool always_noconv() const noexcept;
  int length(stateT&, const externT* from, const externT* end,
             size_t max) const;
  int max_length() const noexcept;

  static locale::id id;

protected:
  ~codecvt();
  virtual result do_out(stateT& state,
                        const internT* from, const internT* from_end, const internT*& from_next,
                        externT* to,         externT* to_end, externT*& to_next) const;
  virtual result do_in(stateT& state,
                       const externT* from, const externT* from_end, const externT*& from_next,
                       internT* to,         internT* to_end, internT*& to_next) const;
  virtual result do_unshift(stateT& state,
                            externT* to,         externT* to_end, externT*& to_next) const;
  virtual int do_encoding() const noexcept;
  virtual bool do_always_noconv() const noexcept;
  virtual int do_length(stateT&, const externT* from,
                        const externT* end, size_t max) const;
  virtual int do_max_length() const noexcept;
};
}
```

1

The class `codecvt<internT,externT,stateT>` is for use when converting from one character encoding to another, such as from wide characters to multibyte characters or between wide character encodings such as Unicode and EUC.

1 The `stateT` argument selects the pair of character encodings being mapped between.

2

The specializations required in Table 81 (22.6.0.1.1) convert the implementation-defined native character set. `codecvt<char, char, mbstate_t>` implements a degenerate conversion; it does not convert at all. The specialization `codecvt<char16_t, char, mbstate_t>` converts between the UTF-16 and UTF-8 encoding forms, and the specialization `codecvt <char32_t, char, mbstate_t>` converts between the UTF-32 and UTF-8 encoding forms. `codecvt<wchar_t,char,mbstate_t>` converts between the native character sets for narrow and wide characters. Specializations on `mbstate_t` perform conversion between encodings known to the library implementer. Other encodings can be converted by specializing on a user-defined `stateT` type. Objects of type `stateT` can contain any state that is useful to communicate to or from the specialized `do_in` or `do_out` members.

22.39.0.0.1 `codecvt` members [locale.codecvt.members]

```
result out(stateT& state,
  const internT* from, const internT* from_end, const internT*& from_next,
        externT* to, externT* to_end, externT*& to_next) const;
```

1 *Returns:* `do_out(state, from, from_end, from_next, to, to_end, to_next)`

```
result unshift(stateT& state,
        externT* to, externT* to_end, externT*& to_next) const;
```

2

Returns: `do_unshift(state, to, to_end, to_next)`

```
result in(stateT& state,
  const externT* from, const externT* from_end, const externT*& from_next,
        internT* to, internT* to_end, internT*& to_next) const;
```

1 *Returns:* `do_in(state, from, from_end, from_next, to, to_end, to_next)`

```
int encoding() const noexcept;
```

2 *Returns:* `do_encoding()`

```
bool always_noconv() const noexcept;
```

3 *Returns:* `do_always_noconv()`

```
int length(stateT& state, const externT* from, const externT* from_end,
           size_t max) const;
```

4 *Returns:* `do_length(state, from,from_end,max)`

```
int max_length() const noexcept;
```

5 *Returns:* `do_max_length()`

22.41.0.0.1 `codecvt` virtual functions [locale.codecvt.virtuals]

```
result do_out(stateT& state,
  const internT* from, const internT* from_end, const internT*& from_next,
  externT* to, externT* to_end, externT*& to_next) const;

result do_in(stateT& state,
  const externT* from, const externT* from_end, const externT*& from_next,
        internT* to, internT* to_end, internT*& to_next) const;
```

1 *Requires:* `(from<=from_end && to<=to_end)` well-defined and `true`; `state` initialized, if at the beginning of a sequence, or else equal to the result of converting the preceding characters in the sequence.

2 *Effects:* Translates characters in the source range `[from,from_end)`, placing the results in sequential positions starting at destination `to`. Converts no more than `(from_end-from)` source elements, and stores no more than `(to_end-to)` destination elements.

Stops if it encounters a character it cannot convert. It always leaves the `from_next` and `to_next` pointers pointing one beyond the last element successfully converted. If returns `noconv`, `internT` and `externT` are the same type and the converted sequence is identical to the input sequence `[from, from_next)`. `to_next` is set equal to `to`, the value of `state` is unchanged, and there are no changes to the values in `[to, to_end)`.

3

A `codecvt` facet that is used by `basic_filebuf` (27.248) shall have the property that if

```
do_out(state, from, from_end, from_next, to, to_end, to_next)
```

would return `ok`, where `from != from_end`, then

```
do_out(state, from, from + 1, from_next, to, to_end, to_next)
```

shall also return `ok`, and that if

```
do_in(state, from, from_end, from_next, to, to_end, to_next)
```

would return `ok`, where `to != to_end`, then

```
do_in(state, from, from_end, from_next, to, to + 1, to_next)
```

shall also return `ok`.[242] [*Note:* As a result of operations on `state`, it can return `ok` or `partial` and set `from_next == from` and `to_next != to`. — *end note*]

1 *Remarks:* Its operations on `state` are unspecified. [*Note:* This argument can be used, for example, to maintain shift state, to specify conversion options (such as count only), or to identify a cache of seek offsets. — *end note*]

2 *Returns:* An enumeration value, as summarized in Table 83.

Table 83 — `do_in/do_out` result values

Value	Meaning
`ok`	completed the conversion
`partial`	not all source characters converted
`error`	encountered a character in `[from,from_end)` that it could not convert
`noconv`	`internT` and `externT` are the same type, and input sequence is identical to converted sequence

A return value of `partial`, if `(from_next==from_end)`, indicates that either the destination sequence has not absorbed all the available destination elements, or that additional source elements are needed before another destination element can be produced.

```
result do_unshift(stateT& state,
  externT* to, externT* to_end, externT*& to_next) const;
```

1 *Requires:* `(to <= to_end)` well defined and true; state initialized, if at the beginning of a sequence, or else equal to the result of converting the preceding characters in the sequence.

2 *Effects:* Places characters starting at `to` that should be appended to terminate a sequence when the current `stateT` is given by `state`.[243] Stores no more than `(to_end-to)` destination elements, and leaves the `to_next` pointer pointing one beyond the last element successfully stored.

3 *Returns:* An enumeration value, as summarized in Table 84.

```
int do_encoding() const noexcept;
```

4 *Returns:*-1 if the encoding of the `externT` sequence is state-dependent; else the constant number of `externT` characters needed to produce an internal character; or 0 if this number is not a constant.[244]

```
bool do_always_noconv() const noexcept;
```

242) Informally, this means that `basic_filebuf` assumes that the mappings from internal to external characters is 1 to N: a `codecvt` facet that is used by `basic_filebuf` must be able to translate characters one internal character at a time.
243) Typically these will be characters to return the state to `stateT()`
244) If `encoding()` yields -1, then more than `max_length()` `externT` elements may be consumed when producing a single `internT` character, and additional `externT` elements may appear at the end of a sequence after those that yield the final `internT` character.

Table 84 — `do_unshift` result values

Value	Meaning
`ok`	completed the sequence
`partial`	space for more than `to_end-to` destination elements was needed to terminate a sequence given the value of `state`
`error`	an unspecified error has occurred
`noconv`	no termination is needed for this `state_type`

5 *Returns:* `true` if `do_in()` and `do_out()` return `noconv` for all valid argument values. `codecvt<char, char, mbstate_t>` returns `true`.

```
int do_length(stateT& state, const externT* from, const externT* from_end,
              size_t max) const;
```

6 *Requires:* `(from<=from_end)` well-defined and `true`; `state` initialized, if at the beginning of a sequence, or else equal to the result of converting the preceding characters in the sequence.

7 *Effects:* The effect on the `state` argument is "as if" it called `do_in(state, from, from_end, from, to, to+max, to)` for `to` pointing to a buffer of at least `max` elements.

8 *Returns:* `(from_next-from)` where `from_next` is the largest value in the range `[from,from_end]` such that the sequence of values in the range `[from,from_next)` represents `max` or fewer valid complete characters of type `internT`. The specialization `codecvt<char, char, mbstate_t>`, returns the lesser of `max` and `(from_end-from)`.

```
int do_max_length() const noexcept;
```

9 *Returns:* The maximum value that `do_length(state, from, from_end, 1)` can return for any valid range `[from, from_end)` and `stateT` value `state`. The specialization `codecvt<char, char, mbstate_t>::do_max_length()` returns 1.

22.44.0.1 Class template `codecvt_byname` [locale.codecvt.byname]

```
namespace std {
  template <class internT, class externT, class stateT>
  class codecvt_byname : public codecvt<internT, externT, stateT> {
  public:
    explicit codecvt_byname(const char*, size_t refs = 0);
    explicit codecvt_byname(const string&, size_t refs = 0);
  protected:
    ~codecvt_byname();
  };
}
```

22.44.1 The numeric category [category.numeric]

1 The classes `num_get<>` and `num_put<>` handle numeric formatting and parsing. Virtual functions are provided for several numeric types. Implementations may (but are not required to) delegate extraction of smaller types to extractors for larger types.[245]

2

245) Parsing `"-1"` correctly into, e.g., an `unsigned short` requires that the corresponding member `get()` at least extract the sign before delegating.

All specifications of member functions for num_put and num_get in the subclauses of 22.44.1 only apply to the specializations required in Tables 81 and 82 (22.6.0.1.1), namely num_get<char>, num_get<wchar_t>, num_get<C, InputIterator>, num_put<char>, num_put<wchar_t>, and num_put<C,OutputIterator>. These specializations refer to the ios_base& argument for formatting specifications (22.31), and to its imbued locale for the numpunct<> facet to identify all numeric punctuation preferences, and also for the ctype<> facet to perform character classification.

1 Extractor and inserter members of the standard iostreams use num_get<> and num_put<> member functions for formatting and parsing numeric values (27.128.0.1.1, 27.198.0.1.1).

22.45.0.1 Class template num_get [locale.num.get]

```
namespace std {
  template <class charT, class InputIterator = istreambuf_iterator<charT> >
  class num_get : public locale::facet {
  public:
    typedef charT            char_type;
    typedef InputIterator    iter_type;

    explicit num_get(size_t refs = 0);

    iter_type get(iter_type in, iter_type end, ios_base&,
                  ios_base::iostate& err, bool& v) const;
    iter_type get(iter_type in, iter_type end, ios_base&,
                  ios_base::iostate& err, long& v) const;
    iter_type get(iter_type in, iter_type end, ios_base&,
                  ios_base::iostate& err, long long& v) const;
    iter_type get(iter_type in, iter_type end, ios_base&,
                  ios_base::iostate& err, unsigned short& v) const;
    iter_type get(iter_type in, iter_type end, ios_base&,
                  ios_base::iostate& err, unsigned int& v) const;
    iter_type get(iter_type in, iter_type end, ios_base&,
                  ios_base::iostate& err, unsigned long& v) const;
    iter_type get(iter_type in, iter_type end, ios_base&,
                  ios_base::iostate& err, unsigned long long& v) const;
    iter_type get(iter_type in, iter_type end, ios_base&,
                  ios_base::iostate& err, float& v) const;
    iter_type get(iter_type in, iter_type end, ios_base&,
                  ios_base::iostate& err, double& v) const;
    iter_type get(iter_type in, iter_type end, ios_base&,
                  ios_base::iostate& err, long double& v) const;
    iter_type get(iter_type in, iter_type end, ios_base&,
                  ios_base::iostate& err, void*& v) const;

    static locale::id id;

  protected:
    ~num_get();
    virtual iter_type do_get(iter_type, iter_type, ios_base&,
                             ios_base::iostate& err, bool& v) const;
    virtual iter_type do_get(iter_type, iter_type, ios_base&,
                             ios_base::iostate& err, long& v) const;
    virtual iter_type do_get(iter_type, iter_type, ios_base&,
                             ios_base::iostate& err, long long& v) const;
    virtual iter_type do_get(iter_type, iter_type, ios_base&,
                             ios_base::iostate& err, unsigned short& v) const;
```

```
    virtual iter_type do_get(iter_type, iter_type, ios_base&,
                             ios_base::iostate& err, unsigned int& v) const;
    virtual iter_type do_get(iter_type, iter_type, ios_base&,
                             ios_base::iostate& err, unsigned long& v) const;
    virtual iter_type do_get(iter_type, iter_type, ios_base&,
                             ios_base::iostate& err, unsigned long long& v) const;
    virtual iter_type do_get(iter_type, iter_type, ios_base&,
                             ios_base::iostate& err, float& v) const;
    virtual iter_type do_get(iter_type, iter_type, ios_base&,
                             ios_base::iostate& err, double& v) const;
    virtual iter_type do_get(iter_type, iter_type, ios_base&,
                             ios_base::iostate& err, long double& v) const;
    virtual iter_type do_get(iter_type, iter_type, ios_base&,
                             ios_base::iostate& err, void*& v) const;
  };
}
```

1 The facet `num_get` is used to parse numeric values from an input sequence such as an istream.

22.45.0.1.1 `num_get` members [facet.num.get.members]

```
iter_type get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, bool& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, long& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, long long& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, unsigned short& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, unsigned int& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, unsigned long& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, unsigned long long& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, float& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, double& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, long double& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, void*& val) const;
```

1 *Returns:* `do_get(in, end, str, err, val)`.

22.45.0.1.2 `num_get` virtual functions [facet.num.get.virtuals]

```
iter_type do_get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, long& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, long long& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, unsigned short& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, unsigned int& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
```

```
  ios_base::iostate& err, unsigned long& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, unsigned long long& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, float& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, double& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, long double& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, void*& val) const;
```

1 *Effects:* Reads characters from `in`, interpreting them according to `str.flags()`, `use_facet<ctype< charT> >(loc)`, and `use_facet< numpunct<charT> >(loc)`, where `loc` is `str.getloc()`.

2

The details of this operation occur in three stages

— Stage 1: Determine a conversion specifier

— Stage 2: Extract characters from `in` and determine a corresponding `char` value for the format expected by the conversion specification determined in stage 1.

— Stage 3: Store results

1

The details of the stages are presented below.

Stage 1: The function initializes local variables via

```
fmtflags flags = str .flags();
fmtflags basefield = (flags & ios_base::basefield);
fmtflags uppercase = (flags & ios_base::uppercase);
fmtflags boolalpha = (flags & ios_base::boolalpha);
```

For conversion to an integral type, the function determines the integral conversion specifier as indicated in Table 85. The table is ordered. That is, the first line whose condition is true applies.

Table 85 — Integer conversions

State	`stdio` equivalent
`basefield == oct`	`%o`
`basefield == hex`	`%X`
`basefield == 0`	`%i`
`signed` integral type	`%d`
`unsigned` integral type	`%u`

For conversions to a floating type the specifier is `%g`.

For conversions to `void*` the specifier is `%p`.

A length modifier is added to the conversion specification, if needed, as indicated in Table 86.

Stage 2: If `in==end` then stage 2 terminates. Otherwise a `charT` is taken from `in` and local variables are initialized as if by

```
char_type ct = *in ;
char c = src[find(atoms, atoms + sizeof(src) - 1, ct) - atoms];
if (ct ==  use_facet<numpunct<charT> >(loc).decimal_point())
c = '.';
bool discard =
  ct == use_facet<numpunct<charT> >(loc).thousands_sep()
  && use_facet<numpunct<charT> >(loc).grouping().length() != 0;
```

Table 86 — Length modifier

Type	Length modifier
short	h
unsigned short	h
long	l
unsigned long	l
long long	ll
unsigned long long	ll
double	l
long double	L

where the values `src` and `atoms` are defined as if by:

```
static const char src[] = "0123456789abcdefxABCDEFX+-";
char_type atoms[sizeof(src)];
use_facet<ctype<charT> >(loc).widen(src, src + sizeof(src), atoms);
```

for this value of `loc`.

If `discard` is true, then if `'.'` has not yet been accumulated, then the position of the character is remembered, but the character is otherwise ignored. Otherwise, if `'.'` has already been accumulated, the character is discarded and Stage 2 terminates. If it is not discarded, then a check is made to determine if `c` is allowed as the next character of an input field of the conversion specifier returned by Stage 1. If so, it is accumulated.

If the character is either discarded or accumulated then `in` is advanced by `++in` and processing returns to the beginning of stage 2.

Stage 3: The sequence of `char`s accumulated in stage 2 (the field) is converted to a numeric value by the rules of one of the functions declared in the header `<cstdlib>`:

— For a signed integer value, the function `strtoll`.

— For an unsigned integer value, the function `strtoull`.

— For a floating-point value, the function `strtold`.

The numeric value to be stored can be one of:

— zero, if the conversion function fails to convert the entire field. `ios_base::failbit` is assigned to `err`.

— the most positive representable value, if the field represents a value too large positive to be represented in `val`. `ios_base::failbit` is assigned to `err`.

— the most negative representable value or zero for an unsigned integer type, if the field represents a value too large negative to be represented in `val`. `ios_base::failbit` is assigned to `err`.

— the converted value, otherwise.

The resultant numeric value is stored in `val`.

1 Digit grouping is checked. That is, the positions of discarded separators is examined for consistency with `use_facet<numpunct<charT> >(loc).grouping()`. If they are not consistent then `ios_base::failbit` is assigned to `err`.

2 In any case, if stage 2 processing was terminated by the test for `in==end` then `err |=ios_base::eofbit` is performed.

```
iter_type do_get(iter_type in, iter_type end, ios_base& str,
                 ios_base::iostate& err, bool& val) const;
```

3

Effects: If `(str.flags()&ios_base::boolalpha)==0` then input proceeds as it would for a `long` except that if a value is being stored into `val`, the value is determined according to the following: If the value to be stored is 0 then `false` is stored. If the value is 1 then `true` is stored. Otherwise `true` is stored and `ios_base::failbit` is assigned to `err`.

1

Otherwise target sequences are determined "as if" by calling the members `falsename()` and `truename()` of the facet obtained by `use_facet<numpunct<charT> >(str.getloc())`. Successive characters in the range `[in,end)` (see 23.224.1) are obtained and matched against corresponding positions in the target sequences only as necessary to identify a unique match. The input iterator `in` is compared to `end` only when necessary to obtain a character. If a target sequence is uniquely matched, `val` is set to the corresponding value. Otherwise `false` is stored and `ios_base::failbit` is assigned to `err`.

1

The `in` iterator is always left pointing one position beyond the last character successfully matched. If `val` is set, then `err` is set to `str.goodbit`; or to `str.eofbit` if, when seeking another character to match, it is found that `(in == end)`. If `val` is not set, then `err` is set to `str.failbit`; or to `(str.failbit|str.eofbit)` if the reason for the failure was that `(in == end)`. [*Example:* For targets `true`: `"a"` and `false`: `"abb"`, the input sequence `"a"` yields `val == true` and `err == str.eofbit`; the input sequence `"abc"` yields `err = str.failbit`, with `in` ending at the `'c'` element. For targets `true`: `"1"` and `false`: `"0"`, the input sequence `"1"` yields `val == true` and `err == str.goodbit`. For empty targets `("")`, any input sequence yields `err == str.failbit`. — *end example*]

1 *Returns:* `in`.

22.90.0.1 Class template `num_put` [locale.nm.put]

```
namespace std {
  template <class charT, class OutputIterator = ostreambuf_iterator<charT> >
  class num_put : public locale::facet {
  public:
    typedef charT            char_type;
    typedef OutputIterator   iter_type;

    explicit num_put(size_t refs = 0);

    iter_type put(iter_type s, ios_base& f, char_type fill, bool v) const;
    iter_type put(iter_type s, ios_base& f, char_type fill, long v) const;
    iter_type put(iter_type s, ios_base& f, char_type fill, long long v) const;
    iter_type put(iter_type s, ios_base& f, char_type fill,
                  unsigned long v) const;
    iter_type put(iter_type s, ios_base& f, char_type fill,
                  unsigned long long v) const;
    iter_type put(iter_type s, ios_base& f, char_type fill,
                  double v) const;
    iter_type put(iter_type s, ios_base& f, char_type fill,
                  long double v) const;
    iter_type put(iter_type s, ios_base& f, char_type fill,
                  const void* v) const;

    static locale::id id;

  protected:
    ~num_put();
```

```
    virtual iter_type do_put(iter_type, ios_base&, char_type fill,
                             bool v) const;
    virtual iter_type do_put(iter_type, ios_base&, char_type fill,
                             long v) const;
    virtual iter_type do_put(iter_type, ios_base&, char_type fill,
                             long long v) const;
    virtual iter_type do_put(iter_type, ios_base&, char_type fill,
                             unsigned long) const;
    virtual iter_type do_put(iter_type, ios_base&, char_type fill,
                             unsigned long long) const;
    virtual iter_type do_put(iter_type, ios_base&, char_type fill,
                             double v) const;
    virtual iter_type do_put(iter_type, ios_base&, char_type fill,
                             long double v) const;
    virtual iter_type do_put(iter_type, ios_base&, char_type fill,
                             const void* v) const;
  };
}
```

1 The facet `num_put` is used to format numeric values to a character sequence such as an ostream.

22.90.0.1.1 `num_put` members [facet.num.put.members]

```
iter_type put(iter_type out, ios_base& str, char_type fill,
  bool val) const;
iter_type put(iter_type out, ios_base& str, char_type fill,
  long val) const;
iter_type put(iter_type out, ios_base& str, char_type fill,
  long long val) const;
iter_type put(iter_type out, ios_base& str, char_type fill,
  unsigned long val) const;
iter_type put(iter_type out, ios_base& str, char_type fill,
  unsigned long long val) const;
iter_type put(iter_type out, ios_base& str, char_type fill,
  double val) const;
iter_type put(iter_type out, ios_base& str, char_type fill,
  long double val) const;
iter_type put(iter_type out, ios_base& str, char_type fill,
  const void* val) const;
```

1 *Returns:* `do_put(out, str, fill, val)`.

22.90.0.1.2 `num_put` virtual functions [facet.num.put.virtuals]

```
iter_type do_put(iter_type out, ios_base& str, char_type fill,
  long val) const;
iter_type do_put(iter_type out, ios_base& str, char_type fill,
  long long val) const;
iter_type do_put(iter_type out, ios_base& str, char_type fill,
  unsigned long val) const;
iter_type do_put(iter_type out, ios_base& str, char_type fill,
  unsigned long long val) const;
iter_type do_put(iter_type out, ios_base& str, char_type fill,
  double val) const;
iter_type do_put(iter_type out, ios_base& str, char_type fill,
  long double val) const;
iter_type do_put(iter_type out, ios_base& str, char_type fill,
```

```
const void* val) const;
```

1 *Effects:* Writes characters to the sequence `out`, formatting `val` as desired. In the following description, a local variable initialized with

```
locale loc = str.getloc();
```

2 The details of this operation occur in several stages:

(2.1) — Stage 1: Determine a printf conversion specifier `spec` and determining the characters that would be printed by `printf` (27.310.1) given this conversion specifier for

```
printf(spec, val )
```

assuming that the current locale is the `"C"` locale.

(2.2) — Stage 2: Adjust the representation by converting each `char` determined by stage 1 to a `charT` using a conversion and values returned by members of `use_facet< numpunct<charT> >(str.getloc())`

(2.3) — Stage 3: Determine where padding is required.

(2.4) — Stage 4: Insert the sequence into the `out`.

3 Detailed descriptions of each stage follow.

4 *Returns:* `out`.

5

Stage 1: The first action of stage 1 is to determine a conversion specifier. The tables that describe this determination use the following local variables

```
fmtflags flags = str.flags() ;
fmtflags basefield =  (flags & (ios_base::basefield));
fmtflags uppercase =  (flags & (ios_base::uppercase));
fmtflags floatfield = (flags & (ios_base::floatfield));
fmtflags showpos =    (flags & (ios_base::showpos));
fmtflags showbase =   (flags & (ios_base::showbase));
```

All tables used in describing stage 1 are ordered. That is, the first line whose condition is true applies. A line without a condition is the default behavior when none of the earlier lines apply.

For conversion from an integral type other than a character type, the function determines the integral conversion specifier as indicated in Table 87.

Table 87 — Integer conversions

State	`stdio` equivalent
`basefield == ios_base::oct`	`%o`
`(basefield == ios_base::hex) && !uppercase`	`%x`
`(basefield == ios_base::hex)`	`%X`
for a `signed` integral type	`%d`
for an `unsigned` integral type	`%u`

For conversion from a floating-point type, the function determines the floating-point conversion specifier as indicated in Table 88.

For conversions from an integral or floating-point type a length modifier is added to the conversion specifier as indicated in Table 89.

The conversion specifier has the following optional additional qualifiers prepended as indicated in Table 90.

Table 88 — Floating-point conversions

State	stdio equivalent
`floatfield == ios_base::fixed`	`%f`
`floatfield == ios_base::scientific && !uppercase`	`%e`
`floatfield == ios_base::scientific`	`%E`
`floatfield == (ios_base::fixed | ios_base::scientific) && !uppercase`	`%a`
`floatfield == (ios_base::fixed | ios_base::scientific)`	`%A`
`!uppercase`	`%g`
otherwise	`%G`

Table 89 — Length modifier

Type	Length modifier
`long`	`l`
`long long`	`ll`
`unsigned long`	`l`
`unsigned long long`	`ll`
`long double`	`L`
otherwise	*none*

Table 90 — Numeric conversions

Type(s)	State	stdio equivalent
an integral type	`flags & showpos`	`+`
	`flags & showbase`	`#`
a floating-point type	`flags & showpos`	`+`
	`flags & showpoint`	`#`

For conversion from a floating-point type, if `floatfield != (ios_base::fixed | ios_base::scientific)`, `str.precision()` is specified as precision in the conversion specification. Otherwise, no precision is specified.

For conversion from `void*` the specifier is `%p`.

The representations at the end of stage 1 consists of the `char`'s that would be printed by a call of `printf(s, val)` where `s` is the conversion specifier determined above.

Stage 2: Any character `c` other than a decimal point(.) is converted to a `charT` via `use_facet<ctype<charT> >(loc).widen( c )`

A local variable `punct` is initialized via

```
const numpunct<charT>& punct = use_facet< numpunct<charT> >(str.getloc());
```

For arithmetic types, `punct.thousands_sep()` characters are inserted into the sequence as determined by the value returned by `punct.do_grouping()` using the method described in 22.102.1.1.2

Decimal point characters(.) are replaced by `punct.decimal_point()`

Stage 3: A local variable is initialized as

```
fmtflags adjustfield=   (flags & (ios_base::adjustfield));
```

The location of any padding[246] is determined according to Table 91.

246) The conversion specification `#o` generates a leading `0` which is *not* a padding character.

Table 91 — Fill padding

State	Location
`adjustfield == ios_base::left`	pad after
`adjustfield == ios_base::right`	pad before
`adjustfield == internal` and a sign occurs in the representation	pad after the sign
`adjustfield == internal` and representation after stage 1 began with 0x or 0X	pad after x or X
otherwise	pad before

If `str.width()` is nonzero and the number of `charT`'s in the sequence after stage 2 is less than `str.width()`, then enough `fill` characters are added to the sequence at the position indicated for padding to bring the length of the sequence to `str.width()`.

`str.width(0)` is called.

Stage 4: The sequence of `charT`'s at the end of stage 3 are output via

```
*out++ = c
```

```
iter_type do_put(iter_type out, ios_base& str, char_type fill,
                 bool val) const;
```

1 *Returns:* If `(str.flags() & ios_base::boolalpha) == 0` returns `do_put(out, str, fill, (int)val)`, otherwise obtains a string `s` as if by

```
string_type s =
  val ? use_facet<numpunct<charT>>(loc).truename()
    : use_facet<numpunct<charT>>(loc).falsename();
```

and then inserts each character `c` of `s` into `out` via `*out++ = c` and returns `out`.

22.102.1 The numeric punctuation facet [facet.numpunct]

22.102.1.1 Class template `numpunct` [locale.numpunct]

```
namespace std {
  template <class charT>
  class numpunct : public locale::facet {
  public:
    typedef charT               char_type;
    typedef basic_string<charT> string_type;

    explicit numpunct(size_t refs = 0);

    char_type    decimal_point()   const;
    char_type    thousands_sep()   const;
    string       grouping()        const;
    string_type  truename()        const;
    string_type  falsename()       const;

    static locale::id id;

  protected:
   ~numpunct();                  // virtual
    virtual char_type    do_decimal_point() const;
```

```
    virtual char_type    do_thousands_sep() const;
    virtual string       do_grouping()      const;
    virtual string_type  do_truename()      const;      // for bool
    virtual string_type  do_falsename()     const;      // for bool
  };
}
```

1 `numpunct<>` specifies numeric punctuation. The specializations required in Table 81 (22.6.0.1.1), namely `numpunct<wchar_t>` and `numpunct<char>`, provide classic `"C"` numeric formats, i.e., they contain information equivalent to that contained in the `"C"` locale or their wide character counterparts as if obtained by a call to `widen`.

2 The syntax for number formats is as follows, where `digit` represents the radix set specified by the `fmtflags` argument value, and `thousands-sep` and `decimal-point` are the results of corresponding `numpunct<charT>` members. Integer values have the format:

```
integer   ::= [sign] units
sign      ::= plusminus
plusminus ::= '+' | '-'
units     ::= digits [thousands-sep units]
digits    ::= digit [digits]
```

and floating-point values have:

```
floatval ::= [sign] units [decimal-point [digits]] [e [sign] digits] |
             [sign]       decimal-point  digits   [e [sign] digits]
e        ::= 'e' | 'E'
```

where the number of digits between `thousands-sep`s is as specified by `do_grouping()`. For parsing, if the `digits` portion contains no thousands-separators, no grouping constraint is applied.

22.102.1.1.1 `numpunct` members [facet.numpunct.members]

`char_type decimal_point() const;`

1 *Returns:* `do_decimal_point()`

`char_type thousands_sep() const;`

2 *Returns:* `do_thousands_sep()`

`string grouping()  const;`

3 *Returns:* `do_grouping()`

```
string_type truename()  const;
string_type falsename() const;
```

4 *Returns:* `do_truename()` or `do_falsename()`, respectively.

22.102.1.1.2 `numpunct` virtual functions [facet.numpunct.virtuals]

`char_type do_decimal_point() const;`

1 *Returns:* A character for use as the decimal radix separator. The required specializations return `'.'` or `L'.'`.

`char_type do_thousands_sep() const;`

2 *Returns:* A character for use as the digit group separator. The required specializations return `','` or `L','`.

```
string do_grouping() const;
```

3 *Returns:* A basic_string<char> `vec` used as a vector of integer values, in which each element `vec[i]` represents the number of digits[247] in the group at position `i`, starting with position 0 as the rightmost group. If `vec.size() <= i`, the number is the same as group `(i-1)`; if `(i<0 || vec[i]<=0 || vec[i]==CHAR_MAX)`, the size of the digit group is unlimited.

4 The required specializations return the empty string, indicating no grouping.

```
string_type do_truename()  const;
string_type do_falsename() const;
```

5 *Returns:* A string representing the name of the boolean value `true` or `false`, respectively.

6 In the base class implementation these names are `"true"` and `"false"`, or `L"true"` and `L"false"`.

22.102.1.2 Class template `numpunct_byname` [locale.numpunct.byname]

```
namespace std {
  template <class charT>
  class numpunct_byname : public numpunct<charT> {
  // this class is specialized for char and wchar_t.
  public:
    typedef charT                char_type;
    typedef basic_string<charT>  string_type;
    explicit numpunct_byname(const char*, size_t refs = 0);
    explicit numpunct_byname(const string&, size_t refs = 0);
  protected:
   ~numpunct_byname();
  };
}
```

22.102.2 The collate category [category.collate]

22.102.2.1 Class template `collate` [locale.collate]

```
namespace std {
  template <class charT>
  class collate : public locale::facet {
  public:
    typedef charT               char_type;
    typedef basic_string<charT> string_type;

    explicit collate(size_t refs = 0);

    int compare(const charT* low1, const charT* high1,
                const charT* low2, const charT* high2) const;
    string_type transform(const charT* low, const charT* high) const;
    long hash(const charT* low, const charT* high) const;

    static locale::id id;

  protected:
    ~collate();
    virtual int do_compare(const charT* low1, const charT* high1,
                           const charT* low2, const charT* high2) const;
```

247) Thus, the string `"\003"` specifies groups of 3 digits each, and `"3"` probably indicates groups of 51 (!) digits each, because 51 is the ASCII value of `"3"`.

```
    virtual string_type do_transform(const charT* low, const charT* high) const;
    virtual long do_hash (const charT* low, const charT* high) const;
  };
}
```

1 The class `collate<charT>` provides features for use in the collation (comparison) and hashing of strings. A locale member function template, `operator()`, uses the collate facet to allow a locale to act directly as the predicate argument for standard algorithms (Clause 25) and containers operating on strings. The specializations required in Table 81 (22.6.0.1.1), namely `collate<char>` and `collate<wchar_t>`, apply lexicographic ordering (25.204.1).

2 Each function compares a string of characters `*p` in the range `[low,high)`.

22.102.2.1.1 `collate` members [locale.collate.members]

```
int compare(const charT* low1, const charT* high1,
            const charT* low2, const charT* high2) const;
```

1 *Returns:* `do_compare(low1, high1, low2, high2)`

```
string_type transform(const charT* low, const charT* high) const;
```

2 *Returns:* `do_transform(low, high)`

```
long hash(const charT* low, const charT* high) const;
```

3 *Returns:* `do_hash(low, high)`

22.102.2.1.2 `collate` virtual functions [locale.collate.virtuals]

```
int do_compare(const charT* low1, const charT* high1,
               const charT* low2, const charT* high2) const;
```

1 *Returns:* `1` if the first string is greater than the second, `-1` if less, zero otherwise. The specializations required in Table 81 (22.6.0.1.1), namely `collate<char>` and `collate<wchar_t>`, implement a lexicographical comparison (25.204.1).

```
string_type do_transform(const charT* low, const charT* high) const;
```

2 *Returns:* A `basic_string<charT>` value that, compared lexicographically with the result of calling `transform()` on another string, yields the same result as calling `do_compare()` on the same two strings.[248]

```
long do_hash(const charT* low, const charT* high) const;
```

3 *Returns:* An integer value equal to the result of calling `hash()` on any other string for which `do_compare()` returns 0 (equal) when passed the two strings. [*Note:* The probability that the result equals that for another string which does not compare equal should be very small, approaching `(1.0/numeric_limits<unsigned long>::max())`. — *end note*]

22.102.2.2 Class template `collate_byname` [locale.collate.byname]

```
namespace std {
  template <class charT>
  class collate_byname : public collate<charT> {
  public:
    typedef basic_string<charT> string_type;
    explicit collate_byname(const char*, size_t refs = 0);
```

248) This function is useful when one string is being compared to many other strings.

```
    explicit collate_byname(const string&, size_t refs = 0);
  protected:
    ~collate_byname();
  };
}
```

22.102.3 The time category [category.time]

1 Templates `time_get<charT,InputIterator>` and `time_put<charT,OutputIterator>` provide date and time formatting and parsing. All specifications of member functions for `time_put` and `time_get` in the subclauses of 22.102.3 only apply to the specializations required in Tables 81 and 82 (22.6.0.1.1). Their members use their `ios_base&`, `ios_base::iostate&`, and `fill` arguments as described in (22.31), and the `ctype<>` facet, to determine formatting details.

22.102.3.1 Class template `time_get` [locale.time.get]

```
namespace std {
  class time_base {
  public:
    enum dateorder { no_order, dmy, mdy, ymd, ydm };
  };

  template <class charT, class InputIterator = istreambuf_iterator<charT> >
  class time_get : public locale::facet, public time_base {
  public:
    typedef charT            char_type;
    typedef InputIterator    iter_type;

    explicit time_get(size_t refs = 0);

    dateorder date_order()  const { return do_date_order(); }
    iter_type get_time(iter_type s, iter_type end, ios_base& f,
                       ios_base::iostate& err, tm* t)  const;
    iter_type get_date(iter_type s, iter_type end, ios_base& f,
                       ios_base::iostate& err, tm* t)  const;
    iter_type get_weekday(iter_type s, iter_type end, ios_base& f,
                       ios_base::iostate& err, tm* t) const;
    iter_type get_monthname(iter_type s, iter_type end, ios_base& f,
                       ios_base::iostate& err, tm* t) const;
    iter_type get_year(iter_type s, iter_type end, ios_base& f,
                       ios_base::iostate& err, tm* t) const;
    iter_type get(iter_type s, iter_type end, ios_base& f,
                       ios_base::iostate& err, tm* t, char format, char modifier = 0) const;
    iter_type get(iter_type s, iter_type end, ios_base& f,
                       ios_base::iostate& err, tm* t, const char_type* fmt,
                       const char_type* fmtend) const;

    static locale::id id;

  protected:
    ~time_get();
    virtual dateorder do_date_order()  const;
    virtual iter_type do_get_time(iter_type s, iter_type end, ios_base&,
                                  ios_base::iostate& err, tm* t) const;
    virtual iter_type do_get_date(iter_type s, iter_type end, ios_base&,
                                  ios_base::iostate& err, tm* t) const;
```

```
    virtual iter_type do_get_weekday(iter_type s, iter_type end, ios_base&,
                                     ios_base::iostate& err, tm* t) const;
    virtual iter_type do_get_monthname(iter_type s, iter_type end, ios_base&,
                                       ios_base::iostate& err, tm* t) const;
    virtual iter_type do_get_year(iter_type s, iter_type end, ios_base&,
                                  ios_base::iostate& err, tm* t) const;
    virtual iter_type do_get(iter_type s, iter_type end, ios_base& f,
                             ios_base::iostate& err, tm* t, char format, char modifier) const;
  };
}
```

1 `time_get` is used to parse a character sequence, extracting components of a time or date into a `struct tm` record. Each `get` member parses a format as produced by a corresponding format specifier to `time_put<>::put`. If the sequence being parsed matches the correct format, the corresponding members of the `struct tm` argument are set to the values used to produce the sequence; otherwise either an error is reported or unspecified values are assigned.[249]

2 If the end iterator is reached during parsing by any of the `get()` member functions, the member sets `ios_base::eofbit` in `err`.

22.102.3.1.1 `time_get` members [locale.time.get.members]

```
dateorder date_order() const;
```

1 *Returns:* `do_date_order()`

```
iter_type get_time(iter_type s, iter_type end, ios_base& str,
                   ios_base::iostate& err, tm* t) const;
```

2 *Returns:* `do_get_time(s, end, str, err, t)`

```
iter_type get_date(iter_type s, iter_type end, ios_base& str,
                   ios_base::iostate& err, tm* t) const;
```

3 *Returns:* `do_get_date(s, end, str, err, t)`

```
iter_type get_weekday(iter_type s, iter_type end, ios_base& str,
                      ios_base::iostate& err, tm* t) const;
iter_type get_monthname(iter_type s, iter_type end, ios_base& str,
                        ios_base::iostate& err, tm* t) const;
```

4 *Returns:* `do_get_weekday(s, end, str, err, t)` or `do_get_monthname(s, end, str, err, t)`

```
iter_type get_year(iter_type s, iter_type end, ios_base& str,
                   ios_base::iostate& err, tm* t) const;
```

5 *Returns:* `do_get_year(s, end, str, err, t)`

```
iter_type get(iter_type s, iter_type end, ios_base& f,
    ios_base::iostate& err, tm* t, char format, char modifier = 0) const;
```

6 *Returns:* `do_get(s, end, f, err, t, format, modifier)`

```
iter_type get(iter_type s, iter_type end, ios_base& f,
    ios_base::iostate& err, tm* t, const char_type* fmt, const char_type* fmtend) const;
```

249) In other words, user confirmation is required for reliable parsing of user-entered dates and times, but machine-generated formats can be parsed reliably. This allows parsers to be aggressive about interpreting user variations on standard formats.

7 *Requires:* `[fmt,fmtend)` shall be a valid range.

8 *Effects:* The function starts by evaluating `err = ios_base::goodbit`. It then enters a loop, reading zero or more characters from `s` at each iteration. Unless otherwise specified below, the loop terminates when the first of the following conditions holds:

— The expression `fmt == fmtend` evaluates to true.

— The expression `err == ios_base::goodbit` evaluates to false.

— The expression `s == end` evaluates to true, in which case the function evaluates `err = ios_base::eofbit | ios_base::failbit`.

— The next element of `fmt` is equal to `'%'`, optionally followed by a modifier character, followed by a conversion specifier character, `format`, together forming a conversion specification valid for the ISO/IEC 9945 function `strptime`. If the number of elements in the range `[fmt,fmtend)` is not sufficient to unambiguously determine whether the conversion specification is complete and valid, the function evaluates `err = ios_base::failbit`. Otherwise, the function evaluates `s = do_get(s, end, f, err, t, format, modifier)`, where the value of `modifier` is `'\0'` when the optional modifier is absent from the conversion specification. If `err == ios_base::goodbit` holds after the evaluation of the expression, the function increments `fmt` to point just past the end of the conversion specification and continues looping.

— The expression `isspace(*fmt, f.getloc())` evaluates to true, in which case the function first increments `fmt` until `fmt == fmtend || !isspace(*fmt, f.getloc())` evaluates to true, then advances `s` until `s == end || !isspace(*s, f.getloc())` is true, and finally resumes looping.

— The next character read from `s` matches the element pointed to by `fmt` in a case-insensitive comparison, in which case the function evaluates `++fmt, ++s` and continues looping. Otherwise, the function evaluates `err = ios_base::failbit`.

1 [*Note:* The function uses the `ctype<charT>` facet installed in `f`'s locale to determine valid whitespace characters. It is unspecified by what means the function performs case-insensitive comparison or whether multi-character sequences are considered while doing so. — *end note*]

2 *Returns:* `s`

22.105.0.0.1 `time_get` virtual functions [locale.time.get.virtuals]

```
dateorder do_date_order() const;
```

1 *Returns:* An enumeration value indicating the preferred order of components for those date formats that are composed of day, month, and year.[250] Returns `no_order` if the date format specified by `'x'` contains other variable components (e.g., Julian day, week number, week day).

```
iter_type do_get_time(iter_type s, iter_type end, ios_base& str,
                      ios_base::iostate& err, tm* t) const;
```

2 *Effects:* Reads characters starting at `s` until it has extracted those `struct tm` members, and remaining format characters, used by `time_put<>::put` to produce the format specified by `"%H:%M:%S"`, or until it encounters an error or end of sequence.

3 *Returns:* An iterator pointing immediately beyond the last character recognized as possibly part of a valid time.

```
iter_type do_get_date(iter_type s, iter_type end, ios_base& str,
                      ios_base::iostate& err, tm* t) const;
```

250) This function is intended as a convenience only, for common formats, and may return `no_order` in valid locales.

4 *Effects:* Reads characters starting at `s` until it has extracted those `struct tm` members and remaining format characters used by `time_put<>::put` to produce one of the following formats, or until it encounters an error. The format depends on the value returned by `date_order()` as shown in Table 92.

Table 92 — `do_get_date` effects

date_order()	Format
no_order	"%m%d%y"
dmy	"%d%m%y"
mdy	"%m%d%y"
ymd	"%y%m%d"
ydm	"%y%d%m"

5 An implementation may also accept additional implementation-defined formats.

6 *Returns:* An iterator pointing immediately beyond the last character recognized as possibly part of a valid date.

```
iter_type do_get_weekday(iter_type s, iter_type end, ios_base& str,
                         ios_base::iostate& err, tm* t) const;
iter_type do_get_monthname(iter_type s, iter_type end, ios_base& str,
                           ios_base::iostate& err, tm* t) const;
```

7 *Effects:* Reads characters starting at `s` until it has extracted the (perhaps abbreviated) name of a weekday or month. If it finds an abbreviation that is followed by characters that could match a full name, it continues reading until it matches the full name or fails. It sets the appropriate `struct tm` member accordingly.

8 *Returns:* An iterator pointing immediately beyond the last character recognized as part of a valid name.

```
iter_type do_get_year(iter_type s, iter_type end, ios_base& str,
                      ios_base::iostate& err, tm* t) const;
```

9 *Effects:* Reads characters starting at `s` until it has extracted an unambiguous year identifier. It is implementation-defined whether two-digit year numbers are accepted, and (if so) what century they are assumed to lie in. Sets the `t->tm_year` member accordingly.

10 *Returns:* An iterator pointing immediately beyond the last character recognized as part of a valid year identifier.

```
iter_type do_get(iter_type s, iter_type end, ios_base& f,
    ios_base::iostate& err, tm* t, char format, char modifier) const;
```

11 *Requires:* `t` shall point to an object.

12 *Effects:* The function starts by evaluating `err = ios_base::goodbit`. It then reads characters starting at `s` until it encounters an error, or until it has extracted and assigned those `struct tm` members, and any remaining format characters, corresponding to a conversion directive appropriate for the ISO/IEC 9945 function `strptime`, formed by concatenating `'%'`, the `modifier` character, when non-NUL, and the `format` character. When the concatenation fails to yield a complete valid directive the function leaves the object pointed to by `t` unchanged and evaluates `err |= ios_base::failbit`. When `s == end` evaluates to true after reading a character the function evaluates `err |= ios_base::eofbit`.

13 For complex conversion directives such as `%c`, `%x`, or `%X`, or directives that involve the optional modifiers `E` or `O`, when the function is unable to unambiguously determine some or all `struct tm` members from

the input sequence `[s,end)`, it evaluates `err |= ios_base::eofbit`. In such cases the values of those `struct tm` members are unspecified and may be outside their valid range.

14 *Remark:* It is unspecified whether multiple calls to `do_get()` with the address of the same `struct tm` object will update the current contents of the object or simply overwrite its members. Portable programs must zero out the object before invoking the function.

15 *Returns:* An iterator pointing immediately beyond the last character recognized as possibly part of a valid input sequence for the given `format` and `modifier`.

22.105.0.1 Class template `time_get_byname` [locale.time.get.byname]

```
namespace std {
  template <class charT, class InputIterator = istreambuf_iterator<charT> >
  class time_get_byname : public time_get<charT, InputIterator> {
  public:
    typedef time_base::dateorder dateorder;
    typedef InputIterator        iter_type;

    explicit time_get_byname(const char*, size_t refs = 0);
    explicit time_get_byname(const string&, size_t refs = 0);
  protected:
    ~time_get_byname();
  };
}
```

22.105.0.2 Class template `time_put` [locale.time.put]

```
namespace std {
  template <class charT, class OutputIterator = ostreambuf_iterator<charT> >
  class time_put : public locale::facet {
  public:
    typedef charT            char_type;
    typedef OutputIterator   iter_type;

    explicit time_put(size_t refs = 0);

    // the following is implemented in terms of other member functions.
    iter_type put(iter_type s, ios_base& f, char_type fill, const tm* tmb,
                  const charT* pattern, const charT* pat_end) const;
    iter_type put(iter_type s, ios_base& f, char_type fill,
                  const tm* tmb, char format, char modifier = 0) const;

    static locale::id id;

  protected:
    ~time_put();
    virtual iter_type do_put(iter_type s, ios_base&, char_type, const tm* t,
                             char format, char modifier) const;
  };
}
```

22.105.0.2.1 `time_put` members [locale.time.put.members]

```
iter_type put(iter_type s, ios_base& str, char_type fill, const tm* t,
              const charT* pattern, const charT* pat_end) const;
iter_type put(iter_type s, ios_base& str, char_type fill, const tm* t,
              char format, char modifier = 0) const;
```

1 *Effects:* The first form steps through the sequence from `pattern` to `pat_end`, identifying characters that are part of a format sequence. Each character that is not part of a format sequence is written to `s` immediately, and each format sequence, as it is identified, results in a call to `do_put`; thus, format elements and other characters are interleaved in the output in the order in which they appear in the pattern. Format sequences are identified by converting each character `c` to a `char` value as if by `ct.narrow(c,0)`, where `ct` is a reference to `ctype<charT>` obtained from `str.getloc()`. The first character of each sequence is equal to `'%'`, followed by an optional modifier character `mod`[251] and a format specifier character `spec` as defined for the function `strftime`. If no modifier character is present, `mod` is zero. For each valid format sequence identified, calls `do_put(s, str, fill, t, spec, mod)`.

2 The second form calls `do_put(s, str, fill, t, format, modifier)`.

3 [*Note:* The `fill` argument may be used in the implementation-defined formats or by derivations. A space character is a reasonable default for this argument. — *end note*]

4 *Returns:* An iterator pointing immediately after the last character produced.

22.105.0.2.2 `time_put` virtual functions [locale.time.put.virtuals]

```
iter_type do_put(iter_type s, ios_base&, char_type fill, const tm* t,
                 char format, char modifier) const;
```

1 *Effects:* Formats the contents of the parameter `t` into characters placed on the output sequence `s`. Formatting is controlled by the parameters `format` and `modifier`, interpreted identically as the format specifiers in the string argument to the standard library function `strftime()`[252], except that the sequence of characters produced for those specifiers that are described as depending on the C locale are instead implementation-defined.[253]

2 *Returns:* An iterator pointing immediately after the last character produced. [*Note:* The `fill` argument may be used in the implementation-defined formats or by derivations. A space character is a reasonable default for this argument. — *end note*]

22.105.0.3 Class template `time_put_byname` [locale.time.put.byname]

```
namespace std {
  template <class charT, class OutputIterator = ostreambuf_iterator<charT> >
  class time_put_byname : public time_put<charT, OutputIterator>
  {
  public:
    typedef charT          char_type;
    typedef OutputIterator iter_type;

    explicit time_put_byname(const char*, size_t refs = 0);
    explicit time_put_byname(const string&, size_t refs = 0);
  protected:
    ~time_put_byname();
  };
}
```

22.105.1 The monetary category [category.monetary]

1 These templates handle monetary formats. A template parameter indicates whether local or international monetary formats are to be used.

251) Although the C programming language defines no modifiers, most vendors do.
252) Interpretation of the `modifier` argument is implementation-defined, but should follow POSIX conventions.
253) Implementations are encouraged to refer to other standards such as POSIX for these definitions.

2 All specifications of member functions for money_put and money_get in the subclauses of 22.105.1 only apply to the specializations required in Tables 81 and 82 (22.6.0.1.1). Their members use their ios_base&, ios_base::iostate&, and fill arguments as described in (22.31), and the moneypunct<> and ctype<> facets, to determine formatting details.

22.105.1.1 Class template money_get [locale.money.get]

```
namespace std {
  template <class charT,
    class InputIterator = istreambuf_iterator<charT> >
  class money_get : public locale::facet {
  public:
    typedef charT               char_type;
    typedef InputIterator       iter_type;
    typedef basic_string<charT> string_type;

    explicit money_get(size_t refs = 0);

    iter_type get(iter_type s, iter_type end, bool intl,
                  ios_base& f, ios_base::iostate& err,
                  long double& units) const;
    iter_type get(iter_type s, iter_type end, bool intl,
                  ios_base& f, ios_base::iostate& err,
                  string_type& digits) const;

    static locale::id id;

  protected:
    ~money_get();
    virtual iter_type do_get(iter_type, iter_type, bool, ios_base&,
                             ios_base::iostate& err, long double& units) const;
    virtual iter_type do_get(iter_type, iter_type, bool, ios_base&,
                             ios_base::iostate& err, string_type& digits) const;
  };
}
```

22.105.1.1.1 money_get members [locale.money.get.members]

```
iter_type get(iter_type s, iter_type end, bool intl,
              ios_base& f, ios_base::iostate& err,
              long double& quant) const;
iter_type get(s, iter_type end, bool intl, ios_base&f,
              ios_base::iostate& err, string_type& quant) const;
```

1 *Returns:* do_get(s, end, intl, f, err, quant)

22.105.1.1.2 money_get virtual functions [locale.money.get.virtuals]

```
iter_type do_get(iter_type s, iter_type end, bool intl,
                 ios_base& str, ios_base::iostate& err,
                 long double& units) const;
iter_type do_get(iter_type s, iter_type end, bool intl,
                 ios_base& str, ios_base::iostate& err,
                 string_type& digits) const;
```

1 *Effects:* Reads characters from s to parse and construct a monetary value according to the format specified by a moneypunct<charT,Intl> facet reference mp and the character mapping specified by a ctype<charT> facet reference ct obtained from the locale returned by str.getloc(),

and `str.flags()`. If a valid sequence is recognized, does not change `err`; otherwise, sets `err` to `(err|str.failbit)`, or `(err|str.failbit|str.eofbit)` if no more characters are available, and does not change `units` or `digits`. Uses the pattern returned by `mp.neg_format()` to parse all values. The result is returned as an integral value stored in `units` or as a sequence of digits possibly preceded by a minus sign (as produced by `ct.widen(c)` where `c` is `'-'` or in the range from `'0'` through `'9'`, inclusive) stored in `digits`. [*Example:* The sequence `$1,056.23` in a common United States locale would yield, for `units`, `105623`, or, for `digits`, `"105623"`. — *end example*] If `mp.grouping()` indicates that no thousands separators are permitted, any such characters are not read, and parsing is terminated at the point where they first appear. Otherwise, thousands separators are optional; if present, they are checked for correct placement only after all format components have been read.

2 Where `money_base::space` or `money_base::none` appears as the last element in the format pattern, no white space is consumed. Otherwise, where `money_base::space` appears in any of the initial elements of the format pattern, at least one white space character is required. Where `money_base::none` appears in any of the initial elements of the format pattern, white space is allowed but not required. If `(str.flags() & str.showbase)` is false, the currency symbol is optional and is consumed only if other characters are needed to complete the format; otherwise, the currency symbol is required.

3 If the first character (if any) in the string `pos` returned by `mp.positive_sign()` or the string `neg` returned by `mp.negative_sign()` is recognized in the position indicated by `sign` in the format pattern, it is consumed and any remaining characters in the string are required after all the other format components. [*Example:* If `showbase` is off, then for a `neg` value of `"()"` and a currency symbol of `"L"`, in `"(100 L)"` the `"L"` is consumed; but if `neg` is `"-"`, the `"L"` in `"-100 L"` is not consumed. — *end example*] If `pos` or `neg` is empty, the sign component is optional, and if no sign is detected, the result is given the sign that corresponds to the source of the empty string. Otherwise, the character in the indicated position must match the first character of `pos` or `neg`, and the result is given the corresponding sign. If the first character of `pos` is equal to the first character of `neg`, or if both strings are empty, the result is given a positive sign.

4 Digits in the numeric monetary component are extracted and placed in `digits`, or into a character buffer `buf1` for conversion to produce a value for `units`, in the order in which they appear, preceded by a minus sign if and only if the result is negative. The value `units` is produced as if by[254]

```
for (int i = 0; i < n; ++i)
  buf2[i] = src[find(atoms, atoms+sizeof(src), buf1[i]) - atoms];
buf2[n] = 0;
sscanf(buf2, "%Lf", &units);
```

where `n` is the number of characters placed in `buf1`, `buf2` is a character buffer, and the values `src` and `atoms` are defined as if by

```
static const char src[] = "0123456789-";
charT atoms[sizeof(src)];
ct.widen(src, src + sizeof(src) - 1, atoms);
```

5 *Returns:* An iterator pointing immediately beyond the last character recognized as part of a valid monetary quantity.

22.105.1.2 Class template `money_put` [locale.money.put]

```
namespace std {
  template <class charT,
    class OutputIterator = ostreambuf_iterator<charT> >
  class money_put : public locale::facet {
```

254) The semantics here are different from `ct.narrow`.

```
  public:
    typedef charT               char_type;
    typedef OutputIterator      iter_type;
    typedef basic_string<charT> string_type;

    explicit money_put(size_t refs = 0);

    iter_type put(iter_type s, bool intl, ios_base& f,
                  char_type fill, long double units) const;
    iter_type put(iter_type s, bool intl, ios_base& f,
                  char_type fill, const string_type& digits) const;

    static locale::id id;

  protected:
    ~money_put();
    virtual iter_type do_put(iter_type, bool, ios_base&, char_type fill,
                             long double units) const;
    virtual iter_type do_put(iter_type, bool, ios_base&, char_type fill,
                             const string_type& digits) const;
  };
}
```

22.105.1.2.1 money_put members [locale.money.put.members]

```
iter_type put(iter_type s, bool intl, ios_base& f, char_type fill,
              long double quant) const;
iter_type put(iter_type s, bool intl, ios_base& f, char_type fill,
              const string_type& quant) const;
```

1 *Returns:* `do_put(s, intl, f, loc, quant)`

22.105.1.2.2 money_put virtual functions [locale.money.put.virtuals]

```
iter_type do_put(iter_type s, bool intl, ios_base& str,
                 char_type fill, long double units) const;
iter_type do_put(iter_type s, bool intl, ios_base& str,
                 char_type fill, const string_type& digits) const;
```

1

Effects: Writes characters to `s` according to the format specified by a `moneypunct<charT,Intl>` facet reference `mp` and the character mapping specified by a `ctype<charT>` facet reference `ct` obtained from the locale returned by `str.getloc()`, and `str.flags()`. The argument `units` is transformed into a sequence of wide characters as if by

```
ct.widen(buf1, buf1 + sprintf(buf1, "%.0Lf", units), buf2)
```

for character buffers `buf1` and `buf2`. If the first character in `digits` or `buf2` is equal to `ct.widen('-')`, then the pattern used for formatting is the result of `mp.neg_format()`; otherwise the pattern is the result of `mp.pos_format()`. Digit characters are written, interspersed with any thousands separators and decimal point specified by the format, in the order they appear (after the optional leading minus sign) in `digits` or `buf2`. In `digits`, only the optional leading minus sign and the immediately subsequent digit characters (as classified according to `ct`) are used; any trailing characters (including digits appearing after a non-digit character) are ignored. Calls `str.width(0)`.

1 *Remarks:* The currency symbol is generated if and only if `(str.flags() & str.showbase)` is nonzero. If the number of characters generated for the specified format is less than the value returned by

`str.width()` on entry to the function, then copies of `fill` are inserted as necessary to pad to the specified width. For the value `af` equal to `(str.flags() & str.adjustfield)`, if `(af == str.internal)` is true, the fill characters are placed where `none` or `space` appears in the formatting pattern; otherwise if `(af == str.left)` is true, they are placed after the other characters; otherwise, they are placed before the other characters. [*Note:* It is possible, with some combinations of format patterns and flag values, to produce output that cannot be parsed using `num_get<>::get`. — *end note*]

2 *Returns:* An iterator pointing immediately after the last character produced.

22.106.0.1 Class template `moneypunct` [locale.moneypunct]

```
namespace std {
  class money_base {
  public:
    enum part { none, space, symbol, sign, value };
    struct pattern { char field[4]; };
  };

  template <class charT, bool International = false>
  class moneypunct : public locale::facet, public money_base {
  public:
    typedef charT char_type;
    typedef basic_string<charT> string_type;

    explicit moneypunct(size_t refs = 0);

    charT        decimal_point() const;
    charT        thousands_sep() const;
    string       grouping()      const;
    string_type  curr_symbol()   const;
    string_type  positive_sign() const;
    string_type  negative_sign() const;
    int          frac_digits()   const;
    pattern      pos_format()    const;
    pattern      neg_format()    const;

    static locale::id id;
    static const bool intl = International;

  protected:
    ~moneypunct();
    virtual charT        do_decimal_point() const;
    virtual charT        do_thousands_sep() const;
    virtual string       do_grouping()      const;
    virtual string_type  do_curr_symbol()   const;
    virtual string_type  do_positive_sign() const;
    virtual string_type  do_negative_sign() const;
    virtual int          do_frac_digits()   const;
    virtual pattern      do_pos_format()    const;
    virtual pattern      do_neg_format()    const;
  };
}
```

1 The `moneypunct<>` facet defines monetary formatting parameters used by `money_get<>` and `money_put<>`. A monetary format is a sequence of four components, specified by a `pattern` value `p`, such that the `part` value

static_cast<part>(p.field[i]) determines the ith component of the format[255] In the field member of a pattern object, each value symbol, sign, value, and either space or none appears exactly once. The value none, if present, is not first; the value space, if present, is neither first nor last.

2 Where none or space appears, white space is permitted in the format, except where none appears at the end, in which case no white space is permitted. The value space indicates that at least one space is required at that position. Where symbol appears, the sequence of characters returned by curr_symbol() is permitted, and can be required. Where sign appears, the first (if any) of the sequence of characters returned by positive_sign() or negative_sign() (respectively as the monetary value is non-negative or negative) is required. Any remaining characters of the sign sequence are required after all other format components. Where value appears, the absolute numeric monetary value is required.

3

The format of the numeric monetary value is a decimal number:

```
value ::= units [ decimal-point [ digits ]] |
  decimal-point digits
```

if frac_digits() returns a positive value, or

```
value ::= units
```

otherwise. The symbol decimal-point indicates the character returned by decimal_point(). The other symbols are defined as follows:

```
units ::= digits [ thousands-sep units ]
digits ::= adigit [ digits ]
```

In the syntax specification, the symbol adigit is any of the values ct.widen(c) for c in the range '0' through '9', inclusive, and ct is a reference of type const ctype<charT>& obtained as described in the definitions of money_get<> and money_put<>. The symbol thousands-sep is the character returned by thousands_sep(). The space character used is the value ct.widen(' '). White space characters are those characters c for which ci.is(space,c) returns true. The number of digits required after the decimal point (if any) is exactly the value returned by frac_digits().

1 The placement of thousands-separator characters (if any) is determined by the value returned by grouping(), defined identically as the member numpunct<>::do_grouping().

22.107.0.0.1 moneypunct members [locale.moneypunct.members]

```
charT        decimal_point() const;
charT        thousands_sep() const;
string       grouping()      const;
string_type  curr_symbol()   const;
string_type  positive_sign() const;
string_type  negative_sign() const;
int          frac_digits()   const;
pattern      pos_format()    const;
pattern      neg_format()    const;
```

1 Each of these functions F returns the result of calling the corresponding virtual member function do_F().

22.107.0.0.2 moneypunct virtual functions [locale.moneypunct.virtuals]

```
charT do_decimal_point() const;
```

1 *Returns:* The radix separator to use in case do_frac_digits() is greater than zero.[256]

255) An array of char, rather than an array of part, is specified for pattern::field purely for efficiency.
256) In common U.S. locales this is '.'.

```
charT do_thousands_sep() const;
```

2 *Returns:* The digit group separator to use in case `do_grouping()` specifies a digit grouping pattern.[257]

```
string do_grouping() const;
```

3 *Returns:* A pattern defined identically as, but not necessarily equal to, the result of `numpunct<charT>::do_grouping()`.[258]

```
string_type do_curr_symbol() const;
```

4 *Returns:* A string to use as the currency identifier symbol.[259]

```
string_type do_positive_sign() const;
string_type do_negative_sign() const;
```

5 *Returns:* `do_positive_sign()` returns the string to use to indicate a positive monetary value;[260] `do_negative_sign()` returns the string to use to indicate a negative value.

```
int do_frac_digits() const;
```

6 *Returns:* The number of digits after the decimal radix separator, if any.[261]

```
pattern do_pos_format() const;
pattern do_neg_format() const;
```

7 *Returns:* The specializations required in Table 82 (22.6.0.1.1), namely `moneypunct<char>`, `moneypunct<wchar_t>`, `moneypunct<char,true>`, and `moneypunct<wchar_t,true>`, return an object of type `pattern` initialized to `{ symbol, sign, none, value }`.[262]

22.107.0.1 Class template `moneypunct_byname` [locale.moneypunct.byname]

```
namespace std {
  template <class charT, bool Intl = false>
  class moneypunct_byname : public moneypunct<charT, Intl> {
  public:
    typedef money_base::pattern pattern;
    typedef basic_string<charT> string_type;

    explicit moneypunct_byname(const char*, size_t refs = 0);
    explicit moneypunct_byname(const string&, size_t refs = 0);
  protected:
    ~moneypunct_byname();
  };
}
```

22.107.1 The message retrieval category [category.messages]

1 Class `messages<charT>` implements retrieval of strings from message catalogs.

257) In common U.S. locales this is `','`.
258) To specify grouping by 3s, the value is `"\003"` *not* `"3"`.
259) For international specializations (second template parameter `true`) this is typically four characters long, usually three letters and a space.
260) This is usually the empty string.
261) In common U.S. locales, this is 2.
262) Note that the international symbol returned by `do_curr_sym()` usually contains a space, itself; for example, `"USD "`.

22.107.1.1 Class template messages [locale.messages]

```
namespace std {
  class messages_base {
  public:
    typedef unspecified signed integer type catalog;
  };

  template <class charT>
  class messages : public locale::facet, public messages_base {
  public:
    typedef charT char_type;
    typedef basic_string<charT> string_type;

    explicit messages(size_t refs = 0);

    catalog open(const basic_string<char>& fn, const locale&) const;
    string_type get(catalog c, int set, int msgid,
                    const string_type& dfault) const;
    void close(catalog c) const;

    static locale::id id;

  protected:
    ~messages();
    virtual catalog do_open(const basic_string<char>&, const locale&) const;
    virtual string_type do_get(catalog, int set, int msgid,
                               const string_type& dfault) const;
    virtual void do_close(catalog) const;
  };
}
```

1 Values of type `messages_base::catalog` usable as arguments to members `get` and `close` can be obtained only by calling member `open`.

22.107.1.1.1 messages members [locale.messages.members]

```
catalog open(const basic_string<char>& name, const locale& loc) const;
```

1 *Returns:* `do_open(name, loc)`.

```
string_type get(catalog cat, int set, int msgid,
                const string_type& dfault) const;
```

2 *Returns:* `do_get(cat, set, msgid, dfault)`.

```
void  close(catalog cat) const;
```

3 *Effects:* Calls `do_close(cat)`.

22.107.1.1.2 messages virtual functions [locale.messages.virtuals]

```
catalog do_open(const basic_string<char>& name,
                const locale& loc) const;
```

1 *Returns:* A value that may be passed to `get()` to retrieve a message from the message catalog identified by the string `name` according to an implementation-defined mapping. The result can be used until it is passed to `close()`.

2 Returns a value less than 0 if no such catalog can be opened.

3 *Remarks:* The locale argument `loc` is used for character set code conversion when retrieving messages, if needed.

```
string_type do_get(catalog cat, int set, int msgid,
              const string_type& dfault) const;
```

4 *Requires:* `cat` shall be a catalog obtained from `open()` and not yet closed.

5 *Returns:* A message identified by arguments `set`, `msgid`, and `dfault`, according to an implementation-defined mapping. If no such message can be found, returns `dfault`.

```
void do_close(catalog cat) const;
```

6 *Requires:* `cat` shall be a catalog obtained from `open()` and not yet closed.

7 *Effects:* Releases unspecified resources associated with `cat`.

8 *Remarks:* The limit on such resources, if any, is implementation-defined.

22.107.1.2 Class template `messages_byname` [locale.messages.byname]

```
namespace std {
  template <class charT>
  class messages_byname : public messages<charT> {
  public:
    typedef messages_base::catalog catalog;
    typedef basic_string<charT>    string_type;

    explicit messages_byname(const char*, size_t refs = 0);
    explicit messages_byname(const string&, size_t refs = 0);
  protected:
    ~messages_byname();
  };
}
```

22.107.2 Program-defined facets [facets.examples]

1 A C++ program may define facets to be added to a locale and used identically as the built-in facets. To create a new facet interface, C++ programs simply derive from `locale::facet` a class containing a static member: `static locale::id id`.

2 [*Note:* The locale member function templates verify its type and storage class. — *end note*]

3 [*Example:* Traditional global localization is still easy:

```
#include <iostream>
#include <locale>
int main(int argc, char** argv) {
  using namespace std;
  locale::global(locale(""));          // set the global locale
                                       // imbue it on all the std streams
  cin.imbue(locale());
  cout.imbue(locale());
  cerr.imbue(locale());
  wcin.imbue(locale());
  wcout.imbue(locale());
  wcerr.imbue(locale());

  return MyObject(argc, argv).doit();
}
```

— *end example*]

4 [*Example:* Greater flexibility is possible:

```
#include <iostream>
#include <locale>
int main() {
  using namespace std;
  cin.imbue(locale(""));        // the user's preferred locale
  cout.imbue(locale::classic());
  double f;
  while (cin >> f) cout << f << endl;
  return (cin.fail() != 0);
}
```

In a European locale, with input `3.456,78`, output is `3456.78`. — *end example*]

5 This can be important even for simple programs, which may need to write a data file in a fixed format, regardless of a user's preference.

6 [*Example:* Here is an example of the use of locales in a library interface.

```
// file: Date.h
#include <iosfwd>
#include <string>
#include <locale>

class Date {
public:
  Date(unsigned day, unsigned month, unsigned year);
  std::string asString(const std::locale& = std::locale());
};

std::istream& operator>>(std::istream& s, Date& d);
std::ostream& operator<<(std::ostream& s, Date d);
```

7 This example illustrates two architectural uses of class `locale`.

8 The first is as a default argument in `Date::asString()`, where the default is the global (presumably user-preferred) locale.

9 The second is in the operators `<<` and `>>`, where a locale "hitchhikes" on another object, in this case a stream, to the point where it is needed.

```
// file: Date.C
#include "Date"                  // includes <ctime>
#include <sstream>
std::string Date::asString(const std::locale& l) {
  using namespace std;
  ostringstream s; s.imbue(l);
  s << *this; return s.str();
}

std::istream& operator>>(std::istream& s, Date& d) {
  using namespace std;
  istream::sentry cerberos(s);
  if (cerberos) {
    ios_base::iostate err = goodbit;
    struct tm t;
```

```
    use_facet< time_get<char> >(s.getloc()).get_date(s, 0, s, err, &t);
    if (!err) d = Date(t.tm_day, t.tm_mon + 1, t.tm_year + 1900);
    s.setstate(err);
  }
  return s;
}
```

— *end example*]

10 A locale object may be extended with a new facet simply by constructing it with an instance of a class derived from `locale::facet`. The only member a C++ program must define is the static member `id`, which identifies your class interface as a new facet.

11 [*Example:* Classifying Japanese characters:

```
// file: <jctype>
#include <locale>
namespace My {
  using namespace std;
  class JCtype : public locale::facet {
  public:
    static locale::id id;        // required for use as a new locale facet
    bool is_kanji (wchar_t c) const;
    JCtype() { }
  protected:
    ~JCtype() { }
  };
}

// file: filt.C
#include <iostream>
#include <locale>
#include "jctype"                // above
std::locale::id My::JCtype::id; // the static JCtype member declared above.

int main() {
  using namespace std;
  typedef ctype<wchar_t> wctype;
  locale loc(locale(""),         // the user's preferred locale ...
         new My::JCtype);        // and a new feature ...
  wchar_t c = use_facet<wctype>(loc).widen('!');
  if (!use_facet<My::JCtype>(loc).is_kanji(c))
    cout << "no it isn't!" << endl;
  return 0;
}
```

12 The new facet is used exactly like the built-in facets. — *end example*]

13 [*Example:* Replacing an existing facet is even easier. The code does not define a member `id` because it is reusing the `numpunct<charT>` facet interface:

```
// file: my_bool.C
#include <iostream>
#include <locale>
#include <string>
namespace My {
  using namespace std;
  typedef numpunct_byname<char> cnumpunct;
```

```
  class BoolNames : public cnumpunct {
  protected:
    string do_truename()  const { return "Oui Oui!"; }
    string do_falsename() const { return "Mais Non!"; }
    ~BoolNames() { }
  public:
    BoolNames(const char* name) : cnumpunct(name) { }
  };
}

int main(int argc, char** argv) {
  using namespace std;
  // make the user's preferred locale, except for...
  locale loc(locale(""), new My::BoolNames(""));
  cout.imbue(loc);
  cout << boolalpha << "Any arguments today? " << (argc > 1) << endl;
  return 0;
}
```

— *end example*]

22.108 Standard code conversion facets [locale.stdcvt]

1 The header `<codecvt>` provides code conversion facets for various character encodings.

2 **Header `<codecvt>` synopsis**

```
namespace std {
  enum codecvt_mode {
    consume_header = 4,
    generate_header = 2,
    little_endian = 1
  };

  template<class Elem, unsigned long Maxcode = 0x10ffff,
    codecvt_mode Mode = (codecvt_mode)0>
  class codecvt_utf8
    : public codecvt<Elem, char, mbstate_t> {
  public:
    explicit codecvt_utf8(size_t refs = 0);
    ~codecvt_utf8();
  };

  template<class Elem, unsigned long Maxcode = 0x10ffff,
    codecvt_mode Mode = (codecvt_mode)0>
  class codecvt_utf16
    : public codecvt<Elem, char, mbstate_t> {
  public:
    explicit codecvt_utf16(size_t refs = 0);
    ~codecvt_utf16();
  };

  template<class Elem, unsigned long Maxcode = 0x10ffff,
    codecvt_mode Mode = (codecvt_mode)0>
  class codecvt_utf8_utf16
    : public codecvt<Elem, char, mbstate_t> {
  public:
```

```
    explicit codecvt_utf8_utf16(size_t refs = 0);
    ~codecvt_utf8_utf16();
  };
}
```

3

For each of the three code conversion facets `codecvt_utf8`, `codecvt_utf16`, and `codecvt_utf8_utf16`:

— `Elem` is the wide-character type, such as `wchar_t`, `char16_t`, or `char32_t`.

— `Maxcode` is the largest wide-character code that the facet will read or write without reporting a conversion error.

— If `(Mode & consume_header)`, the facet shall consume an initial header sequence, if present, when reading a multibyte sequence to determine the endianness of the subsequent multibyte sequence to be read.

— If `(Mode & generate_header)`, the facet shall generate an initial header sequence when writing a multibyte sequence to advertise the endianness of the subsequent multibyte sequence to be written.

— If `(Mode & little_endian)`, the facet shall generate a multibyte sequence in little-endian order, as opposed to the default big-endian order.

1

For the facet `codecvt_utf8`:

— The facet shall convert between UTF-8 multibyte sequences and UCS2 or UCS4 (depending on the size of `Elem`) within the program.

— Endianness shall not affect how multibyte sequences are read or written.

— The multibyte sequences may be written as either a text or a binary file.

1

For the facet `codecvt_utf16`:

— The facet shall convert between UTF-16 multibyte sequences and UCS2 or UCS4 (depending on the size of `Elem`) within the program.

— Multibyte sequences shall be read or written according to the `Mode` flag, as set out above.

— The multibyte sequences may be written only as a binary file. Attempting to write to a text file produces undefined behavior.

1

For the facet `codecvt_utf8_utf16`:

— The facet shall convert between UTF-8 multibyte sequences and UTF-16 (one or two 16-bit codes) within the program.

— Endianness shall not affect how multibyte sequences are read or written.

— The multibyte sequences may be written as either a text or a binary file.

See also: ISO/IEC 10646-1:1993.

Table 93 — Header `<clocale>` synopsis

Type	Name(s)		
Macros:	`LC_ALL`	`LC_COLLATE`	`LC_CTYPE`
	`LC_MONETARY`	`LC_NUMERIC`	`LC_TIME`
	`NULL`		
Struct:	`lconv`		
Functions:	`localeconv`	`setlocale`	

22.118 C library locales [c.locales]

1 Table 93 describes header `<clocale>`.

2 The contents are the same as the Standard C library header `<locale.h>`.

3 Calls to the function `setlocale` may introduce a data race (17.607.0.1) with other calls to `setlocale` or with calls to the functions listed in Table 94.

Table 94 — Potential `setlocale` data races

`fprintf`	`isprint`	`iswdigit`	`localeconv`	`tolower`
`fscanf`	`ispunct`	`iswgraph`	`mblen`	`toupper`
`isalnum`	`isspace`	`iswlower`	`mbstowcs`	`towlower`
`isalpha`	`isupper`	`iswprint`	`mbtowc`	`towupper`
`isblank`	`iswalnum`	`iswpunct`	`setlocale`	`wcscoll`
`iscntrl`	`iswalpha`	`iswspace`	`strcoll`	`wcstod`
`isdigit`	`iswblank`	`iswupper`	`strerror`	`wcstombs`
`isgraph`	`iswcntrl`	`iswxdigit`	`strtod`	`wcsxfrm`
`islower`	`iswctype`	`isxdigit`	`strxfrm`	`wctomb`

See also: ISO C Clause 7.4.

23 Containers library [containers]

23.1 General [containers.general]

1 This Clause describes components that C++ programs may use to organize collections of information.

2 The following subclauses describe container requirements, and components for sequence containers and associative containers, as summarized in Table 95.

Table 95 — Containers library summary

Subclause		Header(s)
23.2	Requirements	

23.540	Sequence containers	`<array>`
		`<deque>`
		`<forward_list>`
		`<list>`
		`<vector>`

23.894	Associative containers	`<map>` `<set>`
23.932	Unordered associative containers	`<unordered_map>` `<unordered_set>`
23.939	Container adaptors	`<queue>` `<stack>`

23.2 Container requirements [container.requirements]

23.2.1 General container requirements [container.requirements.general]

1

Containers are objects that store other objects. They control allocation and deallocation of these objects through constructors, destructors, insert and erase operations.

1

All of the complexity requirements in this Clause are stated solely in terms of the number of operations on the contained objects. [*Example:* the copy constructor of type `vector <vector<int> >` has linear complexity, even though the complexity of copying each contained `vector<int>` is itself linear. — *end example*]

1

For the components affected by this subclause that declare an `allocator_type`, objects stored in these components shall be constructed using the `allocator_traits<allocator_type>::construct` function and destroyed using the `allocator_traits<allocator_type>::destroy` function (20.89.0.1). These functions are called only for the container's element type, not for internal types used by the container. [*Note:* This means, for example, that a node-based container might need to construct nodes containing aligned buffers and call `construct` to place the element into the buffer. — *end note*]

1

In Tables 96, 97, and 98 `X` denotes a container class containing objects of type `T`, `a` and `b` denote values of type `X`, `u` denotes an identifier, `r` denotes a non-const value of type `X`, and `rv` denotes a non-const rvalue of type `X`.

Table 96 — Container requirements

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition	Complexity
`X::value_type`	`T`		*Requires:* `T` is `Erasable` from `X` (see 23.2.1, below)	compile time
`X::reference`	`T&`			compile time
`X::const_reference`	`const T&`			compile time
`X::iterator`	iterator type whose value type is `T`		any iterator category that meets the forward iterator requirements. convertible to `X::const_iterator`.	compile time
`X::const_iterator`	constant iterator type whose value type is `T`		any iterator category that meets the forward iterator requirements.	compile time
`X::difference_type`	signed integer type		is identical to the difference type of `X::iterator` and `X::const_iterator`	compile time
`X::size_type`	unsigned integer type		`size_type` can represent any non-negative value of `difference_type`	compile time
`X u;`			post: `u.empty()`	constant
`X()`			post: `X().empty()`	constant
`X(a)`			*Requires:*`T` is `CopyInsertable` into `X` (see below). post: `a == X(a)`.	linear
`X u(a)` `X u = a;`			*Requires:*`T` is `CopyInsertable` into `X` (see below). post: `u == a`	linear
`X u(rv)` `X u = rv`			post: `u` shall be equal to the value that `rv` had before this construction	(Note B)
`a = rv`	`X&`	All existing elements of `a` are either move assigned to or destroyed	`a` shall be equal to the value that `rv` had before this assignment	linear

Table 96 — Container requirements (continued)

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition	Complexity
`(&a)->~X()`	`void`		note: the destructor is applied to every element of `a`; any memory obtained is deallocated.	linear
`a.begin()`	`iterator`; `const_iterator` for constant `a`			constant
`a.end()`	`iterator`; `const_iterator` for constant `a`			constant
`a.cbegin()`	`const_iterator`	`const_cast<X const&>(a).begin();`		constant
`a.cend()`	`const_iterator`	`const_cast<X const&>(a).end();`		constant
`a == b`	convertible to `bool`	`==` is an equivalence relation. `equal(a.begin(), a.end(), b.begin(), b.end())`	*Requires:* `T` is `EqualityComparable`	Constant if `a.size() != b.size()`, linear otherwise
`a != b`	convertible to `bool`	Equivalent to: `!(a == b)`		linear
`a.swap(b)`	`void`		exchanges the contents of `a` and `b`	(Note A)
`swap(a, b)`	`void`	`a.swap(b)`		(Note A)
`r = a`	`X&`		post: `r == a`.	linear
`a.size()`	`size_type`	`distance(a.begin(), a.end())`		constant
`a.max_size()`	`size_type`	`distance(begin(), end())` for the largest possible container		constant
`a.empty()`	convertible to `bool`	`a.begin() == a.end()`		constant

Notes: the algorithm `equal()` is defined in Clause 25. Those entries marked "(Note A)" or "(Note B)" have linear complexity for `array` and have constant complexity for all other standard containers.

1

The member function `size()` returns the number of elements in the container. The number of elements is defined by the rules of constructors, inserts, and erases.

1

`begin()` returns an iterator referring to the first element in the container. `end()` returns an iterator which is the past-the-end value for the container. If the container is empty, then `begin() == end()`;

1

In the expressions

```
i == j
i != j
i < j
i <= j
i >= j
i > j
i - j
```

where `i` and `j` denote objects of a container's `iterator` type, either or both may be replaced by an object of the container's `const_iterator` type referring to the same element with no change in semantics.

1

Unless otherwise specified, all containers defined in this clause obtain memory using an allocator (see 17.165.0.1). Copy constructors for these container types obtain an allocator by calling `allocator_traits<allocator_type>::select_on_container_copy_construction` on the allocator belonging to the container being copied. Move constructors obtain an allocator by move construction from the allocator belonging to the container being moved. Such move construction of the allocator shall not exit via an exception. All other constructors for these container types take a `const allocator_type&` argument. [*Note:* If an invocation of a constructor uses the default value of an optional allocator argument, then the `Allocator` type must support value initialization. — *end note*] A copy of this allocator is used for any memory allocation performed, by these constructors and by all member functions, during the lifetime of each container object or until the allocator is replaced. The allocator may be replaced only via assignment or `swap()`. Allocator replacement is performed by copy assignment, move assignment, or swapping of the allocator only if `allocator_traits<allocator_type>::propagate_on_container_copy_assignment::value`, `allocator_traits<allocator_type>::propagate_on_container_move_assignment::value`, or `allocator_traits<allocator_type>::propagate_on_container_swap::value` is true within the implementation of the corresponding container operation. The behavior of a call to a container's `swap` function is undefined unless the objects being swapped have allocators that compare equal or `allocator_traits<allocator_type>::propagate_on_container_swap::value` is true. In all container types defined in this Clause, the member `get_allocator()` returns a copy of the allocator used to construct the container or, if that allocator has been replaced, a copy of the most recent replacement.

1

The expression `a.swap(b)`, for containers `a` and `b` of a standard container type other than `array`, shall exchange the values of `a` and `b` without invoking any move, copy, or swap operations on the individual container elements. Any `Compare`, `Pred`, or `Hash` objects belonging to `a` and `b` shall be swappable and shall be exchanged by unqualified calls to non-member `swap`. If `allocator_traits<allocator_type>::propagate_on_container_swap::value` is `true`, then the allocators of `a` and `b` shall also be exchanged using an unqualified call to non-member `swap`. Otherwise, they shall not be swapped, and the behavior is undefined unless `a.get_allocator() == b.get_allocator()`. Every iterator referring to an element in one container before the swap shall refer to the same element in the other container after the swap. It is unspecified whether an iterator with value `a.end()` before the swap will have value `b.end()` after the swap.

1 If the iterator type of a container belongs to the bidirectional or random access iterator categories (24.2), the container is called *reversible* and satisfies the additional requirements in Table 97.

Table 97 — Reversible container requirements

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`X::reverse_iterator`	iterator type whose value type is `T`	`reverse_iterator<iterator>`	compile time
`X::const_reverse_iterator`	constant iterator type whose value type is `T`	`reverse_iterator<const_iterator>`	compile time
`a.rbegin()`	`reverse_iterator`; `const_reverse_iterator` for constant `a`	`reverse_iterator(end())`	constant
`a.rend()`	`reverse_iterator`; `const_reverse_iterator` for constant `a`	`reverse_iterator(begin())`	constant
`a.crbegin()`	`const_reverse_iterator`	`const_cast<X const&>(a).rbegin()`	constant
`a.crend()`	`const_reverse_iterator`	`const_cast<X const&>(a).rend()`	constant

2

Unless otherwise specified (see 23.431.0.1, 23.539.0.1, 23.610.0.1, and 23.806.0.1) all container types defined in this Clause meet the following additional requirements:

— if an exception is thrown by an `insert()` or `emplace()` function while inserting a single element, that function has no effects.

— if an exception is thrown by a `push_back()`, `push_front()`, `emplace_back()`, or `emplace_front()` function, that function has no effects.

— no `erase()`, `clear()`, `pop_back()` or `pop_front()` function throws an exception.

— no copy constructor or assignment operator of a returned iterator throws an exception.

— no `swap()` function throws an exception.

— no `swap()` function invalidates any references, pointers, or iterators referring to the elements of the containers being swapped. [*Note:* The `end()` iterator does not refer to any element, so it may be invalidated. — *end note*]

1

Unless otherwise specified (either explicitly or by defining a function in terms of other functions), invoking a container member function or passing a container as an argument to a library function shall not invalidate iterators to, or change the values of, objects within that container.

1

Table 98 lists operations that are provided for some types of containers but not others. Those containers for which the listed operations are provided shall implement the semantics described in Table 98 unless otherwise stated.

Table 98 — Optional container operations

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition	Complexity
`a < b`	convertible to `bool`	`lexicographical_compare(a.begin(), a.end(), b.begin(), b.end())`	pre: `<` is defined for values of `T`. `<` is a total ordering relationship.	linear
`a > b`	convertible to `bool`	`b < a`		linear
`a <= b`	convertible to `bool`	`!(a > b)`		linear
`a >= b`	convertible to `bool`	`!(a < b)`		linear

Note: the algorithm `lexicographical_compare()` is defined in Clause 25.

1

All of the containers defined in this Clause and in (21.28) except `array` meet the additional requirements of an allocator-aware container, as described in Table 99.

Given a container type `X` having an `allocator_type` identical to `A` and a `value_type` identical to `T` and given an lvalue `m` of type `A`, a pointer `p` of type `T*`, an expression `v` of type (possibly const) `T`, and an rvalue `rv` of type `T`, the following terms are defined. If `X` is not allocator-aware, the terms below are defined as if `A` were `std::allocator<T>` — no allocator object needs to be created and user specializations of `std::allocator<T>` are not instantiated:

— `T` is *`DefaultInsertable` into `X`* means that the following expression is well-formed:

```
allocator_traits<A>::construct(m, p)
```

— An element of `X` is *default-inserted* if it is initialized by evaluation of the expression

```
allocator_traits<A>::construct(m, p)
```

where `p` is the address of the uninitialized storage for the element allocated within `X`.

— `T` is *`MoveInsertable` into `X`* means that the following expression is well-formed:

```
allocator_traits<A>::construct(m, p, rv)
```

and its evaluation causes the following postcondition to hold: The value of `*p` is equivalent to the value of `rv` before the evaluation. [*Note:* rv remains a valid object. Its state is unspecified — *end note*]

— `T` is *`CopyInsertable` into `X`* means that, in addition to `T` being `MoveInsertable` into `X`, the following expression is well-formed:

```
allocator_traits<A>::construct(m, p, v)
```

and its evaluation causes the following postcondition to hold: The value of `v` is unchanged and is equivalent to `*p`.

— `T` is *`EmplaceConstructible` into `X` from `args`*, for zero or more arguments `args`, means that the following expression is well-formed:

```
allocator_traits<A>::construct(m, p, args)
```

— `T` is *`Erasable` from `X`* means that the following expression is well-formed:

```
allocator_traits<A>::destroy(m, p)
```

[*Note:* A container calls `allocator_traits<A>::construct(m, p, args)` to construct an element at `p` using `args`. The default `construct` in `std::allocator` will call `::new((void*)p) T(args)`, but specialized allocators may choose a different definition. — *end note*]

1

In Table 99, `X` denotes an allocator-aware container class with a `value_type` of `T` using allocator of type `A`, `u` denotes a variable, `a` and `b` denote non-const lvalues of type `X`, `t` denotes an lvalue or a const rvalue of type `X`, `rv` denotes a non-const rvalue of type `X`, and `m` is a value of type `A`.

Table 99 — Allocator-aware container requirements

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`allocator_type`	`A`	*Requires:* `allocator_type::value_type` is the same as `X::value_type`.	compile time
`get_allocator()`	`A`		constant
`X()` `X u;`		*Requires:* `A` is `DefaultConstructible`. post: `u.empty()` returns `true`, `u.get_allocator() == A()`	constant
`X(m)` `X u(m);`		post: `u.empty()` returns `true`, `u.get_allocator() == m`	constant
`X(t, m)` `X u(t, m);`		*Requires:* `T` is `CopyInsertable` into `X`. post: `u == t`, `u.get_allocator() == m`	linear
`X(rv)` `X u(rv)`		*Requires:* move construction of `A` shall not exit via an exception. post: `u` shall have the same elements as `rv` had before this construction; the value of `u.get_allocator()` shall be the same as the value of `rv.get_allocator()` before this construction.	constant
`X(rv, m)` `X u(rv, m);`		*Requires:* `T` is `MoveInsertable` into `X`. post: `u` shall have the same elements, or copies of the elements, that `rv` had before this construction, `u.get_allocator() == m`	constant if `m == rv.get_allocator()`, otherwise linear

Table 99 — Allocator-aware container requirements (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`a = t`	`X&`	*Requires:* `T` is `CopyInsertable` into `X` and `CopyAssignable`. post: `a == t`	linear
`a = rv`	`X&`	*Requires:* If `allocator_traits<allocator_type>::propagate_on_container_move_assignment::value` is `false`, `T` is `MoveInsertable` into `X` and `MoveAssignable`. All existing elements of `a` are either move assigned to or destroyed. post: `a` shall be equal to the value that `rv` had before this assignment.	linear
`a.swap(b)`	`void`	exchanges the contents of `a` and `b`	constant

23.184.1 Container data races [container.requirements.dataraces]

1

For purposes of avoiding data races (17.607.0.1), implementations shall consider the following functions to be `const`: `begin`, `end`, `rbegin`, `rend`, `front`, `back`, `data`, `find`, `lower_bound`, `upper_bound`, `equal_range`, `at` and, except in associative or unordered associative containers, `operator[]`.

1

Notwithstanding (17.607.0.1), implementations are required to avoid data races when the contents of the contained object in different elements in the same container, excepting `vector<bool>`, are modified concurrently.

1

[*Note:* For a `vector<int> x` with a size greater than one, `x[1] = 5` and `*x.begin() = 10` can be executed concurrently without a data race, but `x[0] = 5` and `*x.begin() = 10` executed concurrently may result in a data race. As an exception to the general rule, for a `vector<bool> y`, `y[0] = true` may race with `y[1] = true`. — *end note*]

23.224.1 Sequence containers [sequence.reqmts]

1

A sequence container organizes a finite set of objects, all of the same type, into a strictly linear arrangement. The library provides four basic kinds of sequence containers: `vector`, `forward_list`, `list`, and `deque`. In addition, `array` is provided as a sequence container which provides limited sequence operations because it has a fixed number of elements. The library also provides container adaptors that make it easy to construct abstract data types, such as `stack`s or `queue`s, out of the basic sequence container kinds (or out of other kinds of sequence containers that the user might define).

1

The sequence containers offer the programmer different complexity trade-offs and should be used accordingly. `vector` or `array` is the type of sequence container that should be used by default. `list` or `forward_list` should be used when there are frequent insertions and deletions from the middle of the sequence. `deque` is the data structure of choice when most insertions and deletions take place at the beginning or at the end of the sequence.

1

In Tables 100 and 101, `X` denotes a sequence container class, `a` denotes a value of `X` containing elements of type `T`, `A` denotes `X::allocator_type` if it exists and `std::allocator<T>` if it doesn't, `i` and `j` denote iterators satisfying input iterator requirements and refer to elements implicitly convertible to `value_type`, `[i, j)` denotes a valid range, `il` designates an object of type `initializer_list<value_type>`, `n` denotes a value of `X::size_type`, `p` denotes a valid const iterator to `a`, `q` denotes a valid dereferenceable const iterator to `a`, `[q1, q2)` denotes a valid range of const iterators in `a`, `t` denotes an lvalue or a const rvalue of `X::value_type`, and `rv` denotes a non-const rvalue of `X::value_type`. `Args` denotes a template parameter pack; `args` denotes a function parameter pack with the pattern `Args&&`.

1

The complexities of the expressions are sequence dependent.

Table 100 — Sequence container requirements (in addition to container)

Expression	Return type	Assertion/note pre-/post-condition
`X(n, t)` `X a(n, t)`		*Requires:* `T` shall be `CopyInsertable` into `X`. post: `distance(begin(), end()) == n` Constructs a sequence container with `n` copies of `t`
`X(i, j)` `X a(i, j)`		*Requires:* `T` shall be `EmplaceConstructible` into `X` from `*i`. For `vector`, if the iterator does not meet the forward iterator requirements (24.122.1), `T` shall also be `MoveInsertable` into `X`. Each iterator in the range `[i,j)` shall be dereferenced exactly once. post: `distance(begin(), end()) == distance(i, j)` Constructs a sequence container equal to the range `[i, j)`
`X(il);`		Equivalent to `X(il.begin(), il.end())`
`a = il;`	`X&`	*Requires:* `T` is `CopyInsertable` into `X` and `CopyAssignable`. Assigns the range `[il.begin(),il.end())` into `a`. All existing elements of `a` are either assigned to or destroyed. *Returns:* `*this`.

Table 100 — Sequence container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition
`a.emplace(p, args);`	`iterator`	*Requires:* `T` is `EmplaceConstructible` into `X` from `args`. For `vector` and `deque`, T is also `MoveInsertable` into `X` and `MoveAssignable`. *Effects:* Inserts an object of type `T` constructed with `std::forward<Args>(args)...` before `p`.
`a.insert(p,t)`	`iterator`	*Requires:*`T` shall be `CopyInsertable` into `X`. For `vector` and `deque`, `T` shall also be `CopyAssignable`. *Effects:* Inserts a copy of `t` before `p`.
`a.insert(p,rv)`	`iterator`	*Requires:* `T` shall be `MoveInsertable` into `X`. For `vector` and `deque`, `T` shall also be `MoveAssignable`. *Effects:* Inserts a copy of `rv` before `p`.
`a.insert(p,n,t)`	`iterator`	*Requires:*`T` shall be `CopyInsertable` into `X` and `CopyAssignable`. Inserts `n` copies of `t` before `p`.
`a.insert(p,i,j)`	`iterator`	*Requires:*`T` shall be `EmplaceConstructible` into `X` from `*i`. For `vector`, if the iterator does not meet the forward iterator requirements (24.122.1), `T` shall also be `MoveInsertable` into `X` and `MoveAssignable`. Each iterator in the range `[i,j)` shall be dereferenced exactly once. pre: `i` and `j` are not iterators into `a`. Inserts copies of elements in `[i, j)` before `p`
`a.insert(p, il);`	`iterator`	`a.insert(p, il.begin(), il.end())`.
`a.erase(q)`	`iterator`	*Requires:* For `vector` and `deque`, `T` shall be `MoveAssignable`. *Effects:* Erases the element pointed to by `q`
`a.erase(q1,q2)`	`iterator`	*Requires:* For `vector` and `deque`, `T` shall be `MoveAssignable`. *Effects:* Erases the elements in the range `[q1, q2)`.
`a.clear()`	`void`	Destroys all elements in `a`. Invalidates all references, pointers, and iterators referring to the elements of `a` and may invalidate the past-the-end iterator. post: `a.empty()` returns `true`. *Complexity:* Linear.

Table 100 — Sequence container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition
`a.assign(i,j)`	`void`	*Requires:* `T` shall be `EmplaceConstructible` into `X` from `*i` and assignable from `*i`. For `vector`, if the iterator does not meet the forward iterator requirements (24.122.1), `T` shall also be `MoveInsertable` into `X`. Each iterator in the range `[i,j)` shall be dereferenced exactly once. pre: `i`, `j` are not iterators into `a`. Replaces elements in `a` with a copy of `[i, j)`.
`a.assign(il)`	`void`	`a.assign(il.begin(), il.end())`.
`a.assign(n,t)`	`void`	*Requires:* `T` shall be `CopyInsertable` into `X` and `CopyAssignable`. pre: `t` is not a reference into `a`. Replaces elements in `a` with `n` copies of `t`.

1

`iterator` and `const_iterator` types for sequence containers shall be at least of the forward iterator category.

1 The iterator returned from `a.insert(p, t)` points to the copy of `t` inserted into `a`.

2 The iterator returned from `a.insert(p, rv)` points to the copy of `rv` inserted into `a`.

3

The iterator returned from `a.insert(p, n, t)` points to the copy of the first element inserted into `a`, or `p` if `n == 0`.

1 The iterator returned from `a.insert(p, i, j)` points to the copy of the first element inserted into `a`, or `p` if `i == j`.

2 The iterator returned from `a.insert(p, il)` points to the copy of the first element inserted into `a`, or `p` if `il` is empty.

3

The iterator returned from `a.emplace(p, args)` points to the new element constructed from `args` into `a`.

1

The iterator returned from `a.erase(q)` points to the element immediately following `q` prior to the element being erased. If no such element exists, `a.end()` is returned.

1 The iterator returned by `a.erase(q1,q2)` points to the element pointed to by `q2` prior to any elements being erased. If no such element exists, `a.end()` is returned.

2

For every sequence container defined in this Clause and in Clause 21:

— If the constructor

```
template <class InputIterator>
X(InputIterator first, InputIterator last,
  const allocator_type& alloc = allocator_type())
```

is called with a type `InputIterator` that does not qualify as an input iterator, then the constructor shall not participate in overload resolution.

— If the member functions of the forms:

```
template <class InputIterator>          // such as insert()
rt fx1(const_iterator p, InputIterator first, InputIterator last);

template <class InputIterator>          // such as append(), assign()
rt fx2(InputIterator first, InputIterator last);

template <class InputIterator>          // such as replace()
rt fx3(const_iterator i1, const_iterator i2, InputIterator first, InputIterator last);
```

are called with a type `InputIterator` that does not qualify as an input iterator, then these functions shall not participate in overload resolution.

1

The extent to which an implementation determines that a type cannot be an input iterator is unspecified, except that as a minimum integral types shall not qualify as input iterators.

1

Table 101 lists operations that are provided for some types of sequence containers but not others. An implementation shall provide these operations for all container types shown in the "container" column, and shall implement them so as to take amortized constant time.

Table 101 — Optional sequence container operations

Expression	Return type	Operational semantics	Container
`a.front()`	`reference; const_reference` for constant `a`	`*a.begin()`	`basic_string`, `array`, `deque`, `forward_list`, `list`, `vector`

`a.back()`	`reference; const_reference` for constant `a`	`{ auto tmp = a.end();`
		`--tmp;`
		`return *tmp; }`
			`basic_string`, `array`, `deque`, `list`, `vector`

`a.emplace_front(args)`	`void`	Prepends an object of type `T` constructed with `std::forward<Args>(args)....` *Requires:* `T` shall be `EmplaceConstructible` into `X` from `args`.	`deque`, `forward_list`, `list`

Table 101 — Optional sequence container operations (continued)

Expression	Return type	Operational semantics	Container
a.emplace_back(args)	void	Appends an object of type T constructed with std::forward<Args>(args).... *Requires:* T shall be EmplaceConstructible into X from args. For vector, T shall also be MoveInsertable into X.	deque, list, vector
a.push_front(t)	void	Prepends a copy of t. *Requires:* T shall be CopyInsertable into X.	deque, forward_list, list
a.push_front(rv)	void	Prepends a copy of rv. *Requires:* T shall be MoveInsertable into X.	deque, forward_list, list
a.push_back(t)	void	Appends a copy of t. *Requires:* T shall be CopyInsertable into X.	basic_string, deque, list, vector
a.push_back(rv)	void	Appends a copy of rv. *Requires:* T shall be MoveInsertable into X.	basic_string, deque, list, vector
a.pop_front()	void	Destroys the first element. *Requires:* a.empty() shall be false.	deque, forward_list, list
a.pop_back()	void	Destroys the last element. *Requires:* a.empty() shall be false.	basic_string, deque, list, vector
a[n]	reference; const_reference for constant a	*(a.begin() + n)	basic_string, array, deque, vector
a.at(n)	reference; const_reference for constant a	*(a.begin() + n)	basic_string, array, deque, vector

1 The member function at() provides bounds-checked access to container elements. at() throws out_of_range if n >= a.size().

23.294.1 Associative containers [associative.reqmts]

1

Associative containers provide fast retrieval of data based on keys. The library provides four basic kinds of associative containers: `set`, `multiset`, `map` and `multimap`.

1

Each associative container is parameterized on `Key` and an ordering relation `Compare` that induces a strict weak ordering (25.98) on elements of `Key`. In addition, `map` and `multimap` associate an arbitrary *mapped type* `T` with the `Key`. The object of type `Compare` is called the *comparison object* of a container.

1

The phrase "equivalence of keys" means the equivalence relation imposed by the comparison and *not* the `operator==` on keys. That is, two keys `k1` and `k2` are considered to be equivalent if for the comparison object `comp`, `comp(k1, k2) == false && comp(k2, k1) == false`. For any two keys `k1` and `k2` in the same container, calling `comp(k1, k2)` shall always return the same value.

1

An associative container supports *unique keys* if it may contain at most one element for each key. Otherwise, it supports *equivalent keys*. The `set` and `map` classes support unique keys; the `multiset` and `multimap` classes support equivalent keys. For `multiset` and `multimap`, `insert`, `emplace`, and `erase` preserve the relative ordering of equivalent elements.

1

For `set` and `multiset` the value type is the same as the key type. For `map` and `multimap` it is equal to `pair<const Key, T>`.

1

`iterator` of an associative container is of the bidirectional iterator category. For associative containers where the value type is the same as the key type, both `iterator` and `const_iterator` are constant iterators. It is unspecified whether or not `iterator` and `const_iterator` are the same type. [*Note:* `iterator` and `const_iterator` have identical semantics in this case, and `iterator` is convertible to `const_iterator`. Users can avoid violating the One Definition Rule by always using `const_iterator` in their function parameter lists. — *end note*]

1

The associative containers meet all the requirements of Allocator-aware containers (23.2.1), except that for `map` and `multimap`, the requirements placed on `value_type` in Table 96 apply instead to `key_type` and `mapped_type`. [*Note:* For example, in some cases `key_type` and `mapped_type` are required to be `CopyAssignable` even though the associated `value_type`, `pair<const key_type, mapped_type>`, is not `CopyAssignable`. — *end note*]

1

In Table 102, `X` denotes an associative container class, `a` denotes a value of `X`, `a_uniq` denotes a value of `X` when `X` supports unique keys, `a_eq` denotes a value of `X` when `X` supports multiple keys, `a_tran` denotes a value of `X` when the qualified-id `X::key_compare::is_transparent` is valid and denotes a type (14.2114.1), `i` and `j` satisfy input iterator requirements and refer to elements implicitly convertible to `value_type`, `[i,j)` denotes a valid range, `p` denotes a valid const iterator to `a`, `q` denotes a valid dereferenceable const iterator to `a`, `[q1, q2)` denotes a valid range of const iterators in `a`, `il` designates an object of type `initializer_list<value_type>`, `t` denotes a value of `X::value_type`, `k` denotes a value of `X::key_type` and `c` denotes a value of type `X::key_compare`; `kl` is a value such that `a` is partitioned (25.98) with respect to `c(r, kl)`, with `r` the key value of `e` and `e` in `a`; `ku` is a value such that `a` is partitioned with respect to `!c(ku, r)`; `ke` is a value such that `a` is partitioned with respect to `c(r, ke)` and `!c(ke, r)`, with `c(r, ke)` implying `!c(ke, r)`. `A` denotes the storage allocator used by `X`, if any, or `std::allocator<X::value_type>` otherwise, and `m` denotes an allocator of a type convertible to `A`.

Table 102 — Associative container requirements (in addition to container)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`X::key_type`	`Key`		compile time
`mapped_type` (`map` and `multimap` only)	`T`		compile time
`X::value_type` (`set` and `multiset` only)	`Key`	*Requires:* `value_type` is `Erasable` from `X`	compile time
`X::value_type` (`map` and `multimap` only)	`pair<const Key, T>`	*Requires:* `value_type` is `Erasable` from `X`	compile time
`X::key_compare`	`Compare`	defaults to `less<key_type>`	compile time
`X::value_compare`	a binary predicate type	is the same as `key_compare` for `set` and `multiset`; is an ordering relation on pairs induced by the first component (i.e., `Key`) for `map` and `multimap`.	compile time
`X(c)` `X a(c);`		*Requires:* `key_compare` is `CopyConstructible`. *Effects:* Constructs an empty container. Uses a copy of `c` as a comparison object.	constant
`X()` `X a;`		*Requires:* `key_compare` is `DefaultConstructible`. *Effects:* Constructs an empty container. Uses `Compare()` as a comparison object	constant
`X(i,j,c)` `X a(i,j,c);`		*Requires:* `key_compare` is `CopyConstructible`. `value_type` is `EmplaceConstructible` into `X` from `*i`. *Effects:* Constructs an empty container and inserts elements from the range `[i, j)` into it; uses `c` as a comparison object.	$N \log N$ in general (N has the value `distance(i, j)`); linear if `[i, j)` is sorted with `value_comp()`

Table 102 — Associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`X(i,j)` `X a(i,j);`		*Requires:* `key_compare` is `DefaultConstructible`. `value_type` is `EmplaceConstructible` into `X` from `*i`. *Effects:* Same as above, but uses `Compare()` as a comparison object	same as above
`X(il);`		Same as `X(il.begin(), il.end())`.	Same as `X(il.begin(), il.end())`.
`X(il,c);`		Same as `X(il.begin(), il.end(), c)`.	Same as `X(il.begin(), il.end(), c)`.
`a = il`	`X&`	*Requires:* `value_type` is `CopyInsertable` into `X` and `CopyAssignable`. *Effects:* Assigns the range `[il.begin(),il.end())` into `a`. All existing elements of `a` are either assigned to or destroyed.	$NlogN$ in general (where N has the value `il.size() + a.size()`); linear if `[il.begin(),il.end())` is sorted with `value_comp()`.
`a.key_-comp()`	`X::key_-compare`	returns the comparison object out of which `a` was constructed.	constant
`a.value_-comp()`	`X::value_-compare`	returns an object of `value_compare` constructed out of the comparison object	constant
`a_uniq.emplace(args)`	`pair<iterator, bool>`	*Requires:* `value_type` shall be `EmplaceConstructible` into `X` from `args`. *Effects:* Inserts a `value_type` object `t` constructed with `std::forward<Args>(args)...` if and only if there is no element in the container with key equivalent to the key of `t`. The `bool` component of the returned pair is true if and only if the insertion takes place, and the iterator component of the pair points to the element with key equivalent to the key of `t`.	logarithmic

Table 102 — Associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`a_eq.emplace(args)`	`iterator`	*Requires:* `value_type` shall be `EmplaceConstructible` into `X` from `args`. *Effects:* Inserts a `value_type` object `t` constructed with `std::forward<Args>(args)...` and returns the iterator pointing to the newly inserted element. If a range containing elements equivalent to `t` exists in `a_eq`, `t` is inserted at the end of that range.	logarithmic
`a.emplace_hint(p, args)`	`iterator`	equivalent to `a.emplace(std::forward<Args>(args)...)` Return value is an iterator pointing to the element with the key equivalent to the newly inserted element. The element is inserted as close as possible to the position just prior to `p`.	logarithmic in general, but amortized constant if the element is inserted right before `p`
`a_uniq.insert(t)`	`pair<iterator, bool>`	*Requires:* If `t` is a non-const rvalue expression, `value_type` shall be `MoveInsertable` into `X`; otherwise, `value_type` shall be `CopyInsertable` into `X`. *Effects:* Inserts `t` if and only if there is no element in the container with key equivalent to the key of `t`. The `bool` component of the returned pair is true if and only if the insertion takes place, and the `iterator` component of the pair points to the element with key equivalent to the key of `t`.	logarithmic

Table 102 — Associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`a_eq.insert(t)`	`iterator`	*Requires:* If `t` is a non-const rvalue expression, `value_type` shall be `MoveInsertable` into `X`; otherwise, `value_type` shall be `CopyInsertable` into `X`. *Effects:* Inserts `t` and returns the iterator pointing to the newly inserted element. If a range containing elements equivalent to `t` exists in `a_eq`, `t` is inserted at the end of that range.	logarithmic
`a.insert(p, t)`	`iterator`	*Requires:* If `t` is a non-const rvalue expression, `value_type` shall be `MoveInsertable` into `X`; otherwise, `value_type` shall be `CopyInsertable` into `X`. *Effects:* Inserts `t` if and only if there is no element with key equivalent to the key of `t` in containers with unique keys; always inserts `t` in containers with equivalent keys. Always returns the iterator pointing to the element with key equivalent to the key of `t`. `t` is inserted as close as possible to the position just prior to `p`.	logarithmic in general, but amortized constant if `t` is inserted right before `p`.
`a.insert(i, j)`	`void`	*Requires:* `value_type` shall be `EmplaceConstructible` into `X` from `*i`. pre: `i`, `j` are not iterators into `a`. inserts each element from the range `[i,j)` if and only if there is no element with key equivalent to the key of that element in containers with unique keys; always inserts that element in containers with equivalent keys.	$N \log(\text{a.size}() + N)$ (N has the value `distance(i, j)`
`a.insert(il)`	`void`	Equivalent to `a.insert(il.begin(), il.end())`.	

Table 102 — Associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`a.erase(k)`	`size_type`	erases all elements in the container with key equivalent to `k`. returns the number of erased elements.	log(a.size()) + a.count(k)
`a.erase(q)`	`iterator`	erases the element pointed to by `q`. Returns an iterator pointing to the element immediately following `q` prior to the element being erased. If no such element exists, returns `a.end()`.	amortized constant
`a.erase( q1, q2)`	`iterator`	erases all the elements in the range `[q1,q2)`. Returns an iterator pointing to the element pointed to by q2 prior to any elements being erased. If no such element exists, `a.end()` is returned.	log(a.size()) + N where N has the value `distance(q1, q2)`.
`a.clear()`	`void`	`a.erase(a.begin(),a.end())` post: `a.empty()` returns `true`	linear in `a.size()`.
`a.find(k)`	`iterator`; `const_-iterator` for constant `a`.	returns an iterator pointing to an element with the key equivalent to `k`, or `a.end()` if such an element is not found	logarithmic
`a_tran. find(ke)`	`iterator`; `const_-iterator` for constant `a_tran`.	returns an iterator pointing to an element with key `r` such that `!c(r, ke) && !c(ke, r)`, or `a_tran.end()` if such an element is not found	logarithmic
`a.count(k)`	`size_type`	returns the number of elements with key equivalent to `k`	log(a.size()) + a.count(k)
`a_tran. count(ke)`	`size_type`	returns the number of elements with key `r` such that `!c(r, ke) && !c(ke, r)`	log(a_tran.size()) + a_tran.count(ke)
`a.lower_-bound(k)`	`iterator`; `const_-iterator` for constant `a`.	returns an iterator pointing to the first element with key not less than `k`, or `a.end()` if such an element is not found.	logarithmic
`a_tran. lower_-bound(kl)`	`iterator`; `const_-iterator` for constant `a_tran`.	returns an iterator pointing to the first element with key `r` such that `!c(r, kl)`, or `a_tran.end()` if such an element is not found.	logarithmic

Table 102 — Associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`a.upper_bound(k)`	`iterator`; `const_iterator` for constant `a`.	returns an iterator pointing to the first element with key greater than `k`, or `a.end()` if such an element is not found.	logarithmic
`a_tran.upper_bound(ku)`	`iterator`; `const_iterator` for constant `a_tran`.	returns an iterator pointing to the first element with key `r` such that `c(ku, r)`, or `a_tran.end()` if such an element is not found.	logarithmic
`a.equal_range(k)`	`pair<iterator, iterator>`; `pair<const_iterator, const_iterator>` for constant `a`.	equivalent to `make_pair(a.lower_bound(k), a.upper_bound(k))`.	logarithmic
`a_tran.equal_range(ke)`	`pair<iterator, iterator>`; `pair<const_iterator, const_iterator>` for constant `a_tran`.	equivalent to `make_pair(a_tran.lower_bound(ke), a_tran.upper_bound(ke))`.	logarithmic

1

The `insert` and `emplace` members shall not affect the validity of iterators and references to the container, and the `erase` members shall invalidate only iterators and references to the erased elements.

1

The fundamental property of iterators of associative containers is that they iterate through the containers in the non-descending order of keys where non-descending is defined by the comparison that was used to construct them. For any two dereferenceable iterators `i` and `j` such that distance from `i` to `j` is positive,

```
value_comp(*j, *i) == false
```

1

For associative containers with unique keys the stronger condition holds,

```
value_comp(*i, *j) != false.
```

1

When an associative container is constructed by passing a comparison object the container shall not store a pointer or reference to the passed object, even if that object is passed by reference. When an associative container is copied, either through a copy constructor or an assignment operator, the target container shall

then use the comparison object from the container being copied, as if that comparison object had been passed to the target container in its constructor.

1

The member function templates `find`, `count`, `lower_bound`, `upper_bound`, and `equal_range` shall not participate in overload resolution unless the qualified-id `Compare::is_transparent` is valid and denotes a type (14.2114.1).

23.431.0.1 Exception safety guarantees [associative.reqmts.except]

1

For associative containers, no `clear()` function throws an exception. `erase(k)` does not throw an exception unless that exception is thrown by the container's `Compare` object (if any).

1

For associative containers, if an exception is thrown by any operation from within an `insert` or `emplace` function inserting a single element, the insertion has no effect.

1

For associative containers, no `swap` function throws an exception unless that exception is thrown by the swap of the container's `Compare` object (if any).

23.437.1 Unordered associative containers [unord.req]

1

Unordered associative containers provide an ability for fast retrieval of data based on keys. The worst-case complexity for most operations is linear, but the average case is much faster. The library provides four unordered associative containers: `unordered_set`, `unordered_map`, `unordered_multiset`, and `unordered_multimap`.

1 Unordered associative containers conform to the requirements for Containers (23.2), except that the expressions `a == b` and `a != b` have different semantics than for the other container types.

2

Each unordered associative container is parameterized by `Key`, by a function object type `Hash` that meets the `Hash` requirements (17.159.0.1) and acts as a hash function for argument values of type `Key`, and by a binary predicate `Pred` that induces an equivalence relation on values of type `Key`. Additionally, `unordered_map` and `unordered_multimap` associate an arbitrary *mapped type* `T` with the `Key`.

1

The container's object of type `Hash` — denoted by `hash` — is called the *hash function* of the container. The container's object of type `Pred` — denoted by `pred` — is called the *key equality predicate* of the container.

1

Two values `k1` and `k2` of type `Key` are considered equivalent if the container's key equality predicate returns `true` when passed those values. If `k1` and `k2` are equivalent, the container's hash function shall return the same value for both. [*Note:* Thus, when an unordered associative container is instantiated with a non-default `Pred` parameter it usually needs a non-default `Hash` parameter as well. — *end note*] For any two keys `k1` and `k2` in the same container, calling `pred(k1, k2)` shall always return the same value. For any key `k` in a container, calling `hash(k)` shall always return the same value.

1

An unordered associative container supports *unique keys* if it may contain at most one element for each key. Otherwise, it supports *equivalent keys*. `unordered_set` and `unordered_map` support unique keys.

`unordered_multiset` and `unordered_multimap` support equivalent keys. In containers that support equivalent keys, elements with equivalent keys are adjacent to each other in the iteration order of the container. Thus, although the absolute order of elements in an unordered container is not specified, its elements are grouped into *equivalent-key groups* such that all elements of each group have equivalent keys. Mutating operations on unordered containers shall preserve the relative order of elements within each equivalent-key group unless otherwise specified.

1 For `unordered_set` and `unordered_multiset` the value type is the same as the key type. For `unordered_map` and `unordered_multimap` it is `std::pair<const Key, T>`.

2 For unordered containers where the value type is the same as the key type, both `iterator` and `const_iterator` are constant iterators. It is unspecified whether or not `iterator` and `const_iterator` are the same type. [*Note:* `iterator` and `const_iterator` have identical semantics in this case, and `iterator` is convertible to `const_iterator`. Users can avoid violating the One Definition Rule by always using `const_iterator` in their function parameter lists. — *end note*]

3

The elements of an unordered associative container are organized into *buckets*. Keys with the same hash code appear in the same bucket. The number of buckets is automatically increased as elements are added to an unordered associative container, so that the average number of elements per bucket is kept below a bound. Rehashing invalidates iterators, changes ordering between elements, and changes which buckets elements appear in, but does not invalidate pointers or references to elements. For `unordered_multiset` and `unordered_multimap`, rehashing preserves the relative ordering of equivalent elements.

1 The unordered associative containers meet all the requirements of Allocator-aware containers (23.2.1), except that for `unordered_map` and `unordered_multimap`, the requirements placed on `value_type` in Table 96 apply instead to `key_type` and `mapped_type`. [*Note:* For example, `key_type` and `mapped_type` are sometimes required to be `CopyAssignable` even though the associated `value_type`, `pair<const key_type, mapped_type>`, is not `CopyAssignable`. — *end note*]

2

In table 103: `X` is an unordered associative container class, `a` is an object of type `X`, `b` is a possibly const object of type `X`, `a_uniq` is an object of type `X` when `X` supports unique keys, `a_eq` is an object of type `X` when `X` supports equivalent keys, `i` and `j` are input iterators that refer to `value_type`, `[i, j)` is a valid range, `p` and `q2` are valid const iterators to `a`, `q` and `q1` are valid dereferenceable const iterators to `a`, `[q1, q2)` is a valid range in `a`, `il` designates an object of type `initializer_list<value_type>`, `t` is a value of type `X::value_type`, `k` is a value of type `key_type`, `hf` is a possibly const value of type `hasher`, `eq` is a possibly const value of type `key_equal`, `n` is a value of type `size_type`, and `z` is a value of type `float`.

Table 103 — Unordered associative container requirements (in addition to container)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`X::key_type`	`Key`		compile time
`X::mapped_type` (`unordered_map` and `unordered_multimap` only)	`T`		compile time
`X::value_type` (`unordered_set` and `unordered_multiset` only)	`Key`	*Requires:* `value_type` is `Erasable` from `X`	compile time

Table 103 — Unordered associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`X::value_type` (`unordered_map` and `unordered_multimap` only)	`pair<const Key, T>`	*Requires:* `value_type` is `Erasable` from `X`	compile time
`X::hasher`	`Hash`	`Hash` shall be a unary function object type such that the expression `hf(k)` has type `std::size_t`.	compile time
`X::key_equal`	`Pred`	`Pred` shall be a binary predicate that takes two arguments of type `Key`. `Pred` is an equivalence relation.	compile time
`X::local_iterator`	An iterator type whose category, value type, difference type, and pointer and reference types are the same as `X::iterator`'s.	A `local_iterator` object may be used to iterate through a single bucket, but may not be used to iterate across buckets.	compile time
`X::const_local_iterator`	An iterator type whose category, value type, difference type, and pointer and reference types are the same as `X::const_iterator`'s.	A `const_local_iterator` object may be used to iterate through a single bucket, but may not be used to iterate across buckets.	compile time
`X(n, hf, eq)` `X a(n, hf, eq)`	X	*Requires:* `hasher` and `key_equal` are `CopyConstructible`. *Effects:* Constructs an empty container with at least `n` buckets, using `hf` as the hash function and `eq` as the key equality predicate.	$\mathcal{O}(n)$
`X(n, hf)` `X a(n, hf)`	X	*Requires:* `hasher` is `CopyConstructible` and `key_equal` is `DefaultConstructible`. *Effects:* Constructs an empty container with at least `n` buckets, using `hf` as the hash function and `key_equal()` as the key equality predicate.	$\mathcal{O}(n)$

Table 103 — Unordered associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`X(n)` `X a(n)`	X	*Requires:* `hasher` and `key_equal` are `DefaultConstructible`. *Effects:* Constructs an empty container with at least `n` buckets, using `hasher()` as the hash function and `key_equal()` as the key equality predicate.	$\mathcal{O}(n)$
`X()` `X a`	X	*Requires:* `hasher` and `key_equal` are `DefaultConstructible`. *Effects:* Constructs an empty container with an unspecified number of buckets, using `hasher()` as the hash function and `key_equal()` as the key equality predicate.	constant
`X(i, j, n, hf, eq)` `X a(i, j, n, hf, eq)`	X	*Requires:* `hasher` and `key_equal` are `CopyConstructible`. `value_type` is `EmplaceConstructible` into `X` from `*i`. *Effects:* Constructs an empty container with at least `n` buckets, using `hf` as the hash function and `eq` as the key equality predicate, and inserts elements from `[i, j)` into it.	Average case $\mathcal{O}(N)$ (N is `distance(i, j)`), worst case $\mathcal{O}(N^2)$
`X(i, j, n, hf)` `X a(i, j, n, hf)`	X	*Requires:* `hasher` is `CopyConstructible` and `key_equal` is `DefaultConstructible`. `value_type` is `EmplaceConstructible` into `X` from `*i`. *Effects:* Constructs an empty container with at least `n` buckets, using `hf` as the hash function and `key_equal()` as the key equality predicate, and inserts elements from `[i, j)` into it.	Average case $\mathcal{O}(N)$ (N is `distance(i, j)`), worst case $\mathcal{O}(N^2)$

Table 103 — Unordered associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`X(i, j, n)` `X a(i, j, n)`	X	*Requires:* `hasher` and `key_equal` are `DefaultConstructible`. `value_type` is `EmplaceConstructible` into `X` from `*i`. *Effects:* Constructs an empty container with at least `n` buckets, using `hasher()` as the hash function and `key_equal()` as the key equality predicate, and inserts elements from `[i, j)` into it.	Average case $\mathscr{O}(N)$ (N is `distance(i, j)`), worst case $\mathscr{O}(N^2)$
`X(i, j)` `X a(i, j)`	X	*Requires:* `hasher` and `key_equal` are `DefaultConstructible`. `value_type` is `EmplaceConstructible` into `X` from `*i`. *Effects:* Constructs an empty container with an unspecified number of buckets, using `hasher()` as the hash function and `key_equal()` as the key equality predicate, and inserts elements from `[i, j)` into it.	Average case $\mathscr{O}(N)$ (N is `distance(i, j)`), worst case $\mathscr{O}(N^2)$
`X(il)`	`X`	Same as `X(il.begin(), il.end())`.	Same as `X(il.begin(), il.end())`.
`X(il, n)`	`X`	Same as `X(il.begin(), il.end(), n)`.	Same as `X(il.begin(), il.end(), n)`.
`X(il, n, hf)`	`X`	Same as `X(il.begin(), il.end(), n, hf)`.	Same as `X(il.begin(), il.end(), n, hf)`.
`X(il, n, hf, eq)`	`X`	Same as `X(il.begin(), il.end(), n, hf, eq)`.	Same as `X(il.begin(), il.end(), n, hf, eq)`.
`X(b)` `X a(b)`	`X`	Copy constructor. In addition to the requirements of Table 96, copies the hash function, predicate, and maximum load factor.	Average case linear in `b.size()`, worst case quadratic.

Table 103 — Unordered associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`a = b`	`X&`	Copy assignment operator. In addition to the requirements of Table 96, copies the hash function, predicate, and maximum load factor.	Average case linear in `b.size()`, worst case quadratic.
`a = il`	`X&`	*Requires:* `value_type` is `CopyInsertable` into `X` and `CopyAssignable`. *Effects:* Assigns the range `[il.begin(),il.end())` into `a`. All existing elements of `a` are either assigned to or destroyed.	Same as `a = X(il)`.
`b.hash_function()`	`hasher`	Returns `b`'s hash function.	constant
`b.key_eq()`	`key_equal`	Returns `b`'s key equality predicate.	constant
`a_uniq. emplace(args)`	`pair<iterator, bool>`	*Requires:* `value_type` shall be `EmplaceConstructible` into `X` from `args`. *Effects:* Inserts a `value_type` object `t` constructed with `std::forward<Args>(args)...` if and only if there is no element in the container with key equivalent to the key of `t`. The `bool` component of the returned pair is true if and only if the insertion takes place, and the iterator component of the pair points to the element with key equivalent to the key of `t`.	Average case $\mathcal{O}(1)$, worst case $\mathcal{O}$(`a_uniq. size()`).
`a_eq.emplace(args)`	`iterator`	*Requires:* `value_type` shall be `EmplaceConstructible` into `X` from `args`. *Effects:* Inserts a `value_type` object `t` constructed with `std::forward<Args>(args)...` and returns the iterator pointing to the newly inserted element.	Average case $\mathcal{O}(1)$, worst case $\mathcal{O}$(`a_eq. size()`).

Table 103 — Unordered associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`a.emplace_hint(p, args)`	`iterator`	*Requires:* `value_type` shall be `EmplaceConstructible` into `X` from `args`. *Effects:* Equivalent to `a.emplace( std::forward<Args>(args)...)`. Return value is an iterator pointing to the element with the key equivalent to the newly inserted element. The `const_iterator p` is a hint pointing to where the search should start. Implementations are permitted to ignore the hint.	Average case $\mathcal{O}(1)$, worst case $\mathcal{O}$(`a.size()`).
`a_uniq.insert(t)`	`pair<iterator, bool>`	*Requires:* If `t` is a non-const rvalue expression, `value_type` shall be `MoveInsertable` into `X`; otherwise, `value_type` shall be `CopyInsertable` into `X`. *Effects:* Inserts `t` if and only if there is no element in the container with key equivalent to the key of `t`. The `bool` component of the returned pair indicates whether the insertion takes place, and the `iterator` component points to the element with key equivalent to the key of `t`.	Average case $\mathcal{O}(1)$, worst case $\mathcal{O}$(`a_uniq.size()`).
`a_eq.insert(t)`	`iterator`	*Requires:* If `t` is a non-const rvalue expression, `value_type` shall be `MoveInsertable` into `X`; otherwise, `value_type` shall be `CopyInsertable` into `X`. *Effects:* Inserts `t`, and returns an iterator pointing to the newly inserted element.	Average case $\mathcal{O}(1)$, worst case $\mathcal{O}$(`a_eq.size()`).

Table 103 — Unordered associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`a.insert(q, t)`	`iterator`	*Requires:* If `t` is a non-const rvalue expression, `value_type` shall be `MoveInsertable` into `X`; otherwise, `value_type` shall be `CopyInsertable` into `X`. *Effects:* Equivalent to a.insert(t). Return value is an iterator pointing to the element with the key equivalent to that of `t`. The iterator `q` is a hint pointing to where the search should start. Implementations are permitted to ignore the hint.	Average case $\mathcal{O}(1)$, worst case $\mathcal{O}$(`a.size()`).
`a.insert(i, j)`	`void`	*Requires:* `value_type` shall be `EmplaceConstructible` into `X` from `*i`. Pre: `i` and `j` are not iterators in `a`. Equivalent to `a.insert(t)` for each element in `[i,j)`.	Average case $\mathcal{O}(N)$, where N is `distance(i, j)`. Worst case $\mathcal{O}(N *$ `(a.size()) + N`).
`a.insert(il)`	`void`	Same as `a.insert(il.begin(), il.end())`.	Same as `a.insert( il.begin(), il.end())`.
`a.erase(k)`	`size_type`	Erases all elements with key equivalent to `k`. Returns the number of elements erased.	Average case $\mathcal{O}$(`a.count(k)`). Worst case $\mathcal{O}$(`a.size()`).
`a.erase(q)`	`iterator`	Erases the element pointed to by `q`. Return value is the iterator immediately following `q` prior to the erasure.	Average case $\mathcal{O}(1)$, worst case $\mathcal{O}$(`a.size()`).
`a.erase(q1, q2)`	`iterator`	Erases all elements in the range `[q1, q2)`. Return value is the iterator immediately following the erased elements prior to the erasure.	Average case linear in `distance(q1, q2)`, worst case $\mathcal{O}$(`a.size()`).
`a.clear()`	`void`	Erases all elements in the container. Post: `a.empty()` returns `true`	Linear.
`b.find(k)`	`iterator`; `const_iterator` for const `b`.	Returns an iterator pointing to an element with key equivalent to `k`, or `b.end()` if no such element exists.	Average case $\mathcal{O}(1)$, worst case $\mathcal{O}$(`b.size()`).

Table 103 — Unordered associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`b.count(k)`	`size_type`	Returns the number of elements with key equivalent to `k`.	Average case $\mathcal{O}$(`b.count(k)`), worst case $\mathcal{O}$(`b.size()`).
`b.equal_range(k)`	`pair<iterator, iterator>`; `pair<const_iterator, const_iterator>` for const `b`.	Returns a range containing all elements with keys equivalent to `k`. Returns `make_pair(b.end(), b.end())` if no such elements exist.	Average case $\mathcal{O}$(`b.count(k)`). Worst case $\mathcal{O}$(`b.size()`).
`b.bucket_count()`	`size_type`	Returns the number of buckets that `b` contains.	Constant
`b.max_bucket_count()`	`size_type`	Returns an upper bound on the number of buckets that `b` might ever contain.	Constant
`b.bucket(k)`	`size_type`	Pre: `b.bucket_count() > 0`. Returns the index of the bucket in which elements with keys equivalent to `k` would be found, if any such element existed. Post: the return value shall be in the range `[0, b.bucket_count())`.	Constant
`b.bucket_size(n)`	`size_type`	Pre: `n` shall be in the range `[0, b.bucket_count())`. Returns the number of elements in the `n` th bucket.	$\mathcal{O}$(`b.bucket_size(n)`)
`b.begin(n)`	`local_iterator`; `const_local_iterator` for const `b`.	Pre: `n` shall be in the range `[0, b.bucket_count())`. `b.begin(n)` returns an iterator referring to the first element in the bucket. If the bucket is empty, then `b.begin(n) == b.end(n)`.	Constant
`b.end(n)`	`local_iterator`; `const_local_iterator` for const `b`.	Pre: `n` shall be in the range `[0, b.bucket_count())`. `b.end(n)` returns an iterator which is the past-the-end value for the bucket.	Constant
`b.cbegin(n)`	`const_local_iterator`	Pre: `n` shall be in the range `[0, b.bucket_count())`. Note: `[b.cbegin(n), b.cend(n))` is a valid range containing all of the elements in the `n` th bucket.	Constant
`b.cend(n)`	`const_local_iterator`	Pre: `n` shall be in the range `[0, b.bucket_count())`.	Constant

Table 103 — Unordered associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`b.load_factor()`	`float`	Returns the average number of elements per bucket.	Constant
`b.max_load_factor()`	`float`	Returns a positive number that the container attempts to keep the load factor less than or equal to. The container automatically increases the number of buckets as necessary to keep the load factor below this number.	Constant
`a.max_load_-factor(z)`	`void`	Pre: `z` shall be positive. May change the container's maximum load factor, using `z` as a hint.	Constant
`a.rehash(n)`	`void`	Post: `a.bucket_count() > a.size() / a.max_load_factor()` and `a.bucket_count() >= n`.	Average case linear in `a.size()`, worst case quadratic.
`a.reserve(n)`	`void`	Same as `a.rehash(ceil(n / a.max_load_factor()))`.	Average case linear in `a.size()`, worst case quadratic.

1

Two unordered containers `a` and `b` compare equal if `a.size() == b.size()` and, for every equivalent-key group `[Ea1,Ea2)` obtained from `a.equal_range(Ea1)`, there exists an equivalent-key group `[Eb1,Eb2)` obtained from `b.equal_range(Ea1)`, such that `is_permutation(Ea1, Ea2, Eb1, Eb2)` returns `true`. For `unordered_set` and `unordered_map`, the complexity of `operator==` (i.e., the number of calls to the `==` operator of the `value_type`, to the predicate returned by `key_equal()`, and to the hasher returned by `hash_function()`) is proportional to N in the average case and to N^2 in the worst case, where N is a.size(). For `unordered_multiset` and `unordered_multimap`, the complexity of `operator==` is proportional to $\sum E_i^2$ in the average case and to N^2 in the worst case, where N is `a.size()`, and E_i is the size of the ith equivalent-key group in `a`. However, if the respective elements of each corresponding pair of equivalent-key groups Ea_i and Eb_i are arranged in the same order (as is commonly the case, e.g., if `a` and `b` are unmodified copies of the same container), then the average-case complexity for `unordered_multiset` and `unordered_multimap` becomes proportional to N (but worst-case complexity remains $\mathscr{O}(N^2)$, e.g., for a pathologically bad hash function). The behavior of a program that uses `operator==` or `operator!=` on unordered containers is undefined unless the `Hash` and `Pred` function objects respectively have the same behavior for both containers and the equality comparison operator for `Key` is a refinement[263] of the partition into equivalent-key groups produced by `Pred`.

1 The iterator types `iterator` and `const_iterator` of an unordered associative container are of at least the forward iterator category. For unordered associative containers where the key type and value type are the

263) Equality comparison is a refinement of partitioning if no two objects that compare equal fall into different partitions.

same, both `iterator` and `const_iterator` are const iterators.

2

The `insert` and `emplace` members shall not affect the validity of references to container elements, but may invalidate all iterators to the container. The `erase` members shall invalidate only iterators and references to the erased elements, and preserve the relative order of the elements that are not erased.

1

The `insert` and `emplace` members shall not affect the validity of iterators if `(N+n) < z * B`, where `N` is the number of elements in the container prior to the insert operation, `n` is the number of elements inserted, `B` is the container's bucket count, and `z` is the container's maximum load factor.

23.539.0.1 Exception safety guarantees [unord.req.except]

1 For unordered associative containers, no `clear()` function throws an exception. `erase(k)` does not throw an exception unless that exception is thrown by the container's `Hash` or `Pred` object (if any).

2 For unordered associative containers, if an exception is thrown by any operation other than the container's hash function from within an `insert` or `emplace` function inserting a single element, the insertion has no effect.

3 For unordered associative containers, no `swap` function throws an exception unless that exception is thrown by the swap of the container's Hash or Pred object (if any).

4 For unordered associative containers, if an exception is thrown from within a `rehash()` function other than by the container's hash function or comparison function, the `rehash()` function has no effect.

23.540 Sequence containers [sequences]

23.540.1 In general [sequences.general]

1

The headers `<array>`, `<deque>`, `<forward_list>`, `<list>`, and `<vector>` define template classes that meet the requirements for sequence containers.

1

The headers `<queue>` and `<stack>` define container adaptors (23.939) that also meet the requirements for sequence containers.

Header `<array>` synopsis

```
#include <initializer_list>

namespace std {
  template <class T, size_t N> struct array;
  template <class T, size_t N>
    bool operator==(const array<T,N>& x, const array<T,N>& y);
  template <class T, size_t N>
    bool operator!=(const array<T,N>& x, const array<T,N>& y);
  template <class T, size_t N>
    bool operator<(const array<T,N>& x, const array<T,N>& y);
  template <class T, size_t N>
    bool operator>(const array<T,N>& x, const array<T,N>& y);
  template <class T, size_t N>
    bool operator<=(const array<T,N>& x, const array<T,N>& y);
  template <class T, size_t N>
    bool operator>=(const array<T,N>& x, const array<T,N>& y);
  template <class T, size_t N>
```

```
    void swap(array<T,N>& x, array<T,N>& y) noexcept(noexcept(x.swap(y)));

  template <class T> class tuple_size;
  template <size_t I, class T> class tuple_element;
  template <class T, size_t N>
    struct tuple_size<array<T, N> >;
  template <size_t I, class T, size_t N>
    struct tuple_element<I, array<T, N> >;
  template <size_t I, class T, size_t N>
    constexpr T& get(array<T, N>&) noexcept;
  template <size_t I, class T, size_t N>
    constexpr T&& get(array<T, N>&&) noexcept;
  template <size_t I, class T, size_t N>
    constexpr const T& get(const array<T, N>&) noexcept;
}
```

Header <deque> synopsis

```
#include <initializer_list>

namespace std {
  template <class T, class Allocator = allocator<T> > class deque;
  template <class T, class Allocator>
    bool operator==(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator<(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator!=(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator>(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator>=(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator<=(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
  template <class T, class Allocator>
    void swap(deque<T,Allocator>& x, deque<T,Allocator>& y);
}
```

Header <forward_list> synopsis

```
#include <initializer_list>

namespace std {
  template <class T, class Allocator = allocator<T> > class forward_list;
  template <class T, class Allocator>
    bool operator==(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator< (const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator!=(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator> (const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator>=(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator<=(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
```

```
  template <class T, class Allocator>
    void swap(forward_list<T,Allocator>& x, forward_list<T,Allocator>& y);
}
```

Header <list> synopsis

```
#include <initializer_list>

namespace std {
  template <class T, class Allocator = allocator<T> > class list;
  template <class T, class Allocator>
    bool operator==(const list<T,Allocator>& x, const list<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator< (const list<T,Allocator>& x, const list<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator!=(const list<T,Allocator>& x, const list<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator> (const list<T,Allocator>& x, const list<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator>=(const list<T,Allocator>& x, const list<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator<=(const list<T,Allocator>& x, const list<T,Allocator>& y);
  template <class T, class Allocator>
    void swap(list<T,Allocator>& x, list<T,Allocator>& y);
}
```

Header <vector> synopsis

```
#include <initializer_list>

namespace std {
  template <class T, class Allocator = allocator<T> > class vector;
  template <class T, class Allocator>
    bool operator==(const vector<T,Allocator>& x,const vector<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator< (const vector<T,Allocator>& x,const vector<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator!=(const vector<T,Allocator>& x,const vector<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator> (const vector<T,Allocator>& x,const vector<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator>=(const vector<T,Allocator>& x,const vector<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator<=(const vector<T,Allocator>& x,const vector<T,Allocator>& y);
  template <class T, class Allocator>
    void swap(vector<T,Allocator>& x, vector<T,Allocator>& y);

  template <class Allocator> class vector<bool,Allocator>;

  // hash support
  template <class T> struct hash;
  template <class Allocator> struct hash<vector<bool, Allocator> >;
}
```

23.542.1 Class template array [array]

23.542.1.1 Class template array overview [array.overview]

1

The header `<array>` defines a class template for storing fixed-size sequences of objects. An `array` supports random access iterators. An instance of `array<T, N>` stores `N` elements of type `T`, so that `size() == N` is an invariant. The elements of an `array` are stored contiguously, meaning that if `a` is an `array<T, N>` then it obeys the identity `&a[n] == &a[0] + n` for all `0 <= n < N`.

1

An `array` is an aggregate (8.1817.1) that can be initialized with the syntax

```
array<T, N> a = { initializer-list };
```

where *initializer-list* is a comma-separated list of up to `N` elements whose types are convertible to `T`.

1

An `array` satisfies all of the requirements of a container and of a reversible container (23.2), except that a default constructed `array` object is not empty and that `swap` does not have constant complexity. An `array` satisfies some of the requirements of a sequence container (23.224.1). Descriptions are provided here only for operations on `array` that are not described in one of these tables and for operations where there is additional semantic information.

```
namespace std {
  template <class T, size_t N>
  struct array {
    // types:
    typedef T&                                    reference;
    typedef const T&                              const_reference;
    typedef implementation-defined                iterator;
    typedef implementation-defined                const_iterator;
    typedef size_t                                size_type;
    typedef ptrdiff_t                             difference_type;
    typedef T                                     value_type;
    typedef T*                                    pointer;
    typedef const T*                              const_pointer;
    typedef std::reverse_iterator<iterator>       reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

    T       elems[N];            // exposition only

    // no explicit construct/copy/destroy for aggregate type

    void fill(const T& u);
    void swap(array&) noexcept(noexcept(swap(declval<T&>(), declval<T&>())));

    // iterators:
    iterator               begin() noexcept;
    const_iterator         begin() const noexcept;
    iterator               end() noexcept;
    const_iterator         end() const noexcept;

    reverse_iterator       rbegin() noexcept;
    const_reverse_iterator rbegin() const noexcept;
    reverse_iterator       rend() noexcept;
    const_reverse_iterator rend() const noexcept;
```

```
    const_iterator         cbegin() const noexcept;
    const_iterator         cend() const noexcept;
    const_reverse_iterator crbegin() const noexcept;
    const_reverse_iterator crend() const noexcept;

    // capacity:
    constexpr size_type size() const noexcept;
    constexpr size_type max_size() const noexcept;
    constexpr bool      empty() const noexcept;

    // element access:
    reference                 operator[](size_type n);
    constexpr const_reference operator[](size_type n) const;
    reference                 at(size_type n);
    constexpr const_reference at(size_type n) const;
    reference                 front();
    constexpr const_reference front() const;
    reference                 back();
    constexpr const_reference back() const;

    T *       data() noexcept;
    const T * data() const noexcept;
  };
}
```

1

[*Note:* The member variable `elems` is shown for exposition only, to emphasize that `array` is a class aggregate. The name `elems` is not part of `array`'s interface. — *end note*]

23.579.0.1 `array` constructors, copy, and assignment [array.cons]

1

The conditions for an aggregate (8.1817.1) shall be met. Class `array` relies on the implicitly-declared special member functions (12.30, 12.807, and 12.1577) to conform to the container requirements table in 23.2. In addition to the requirements specified in the container requirements table, the implicit move constructor and move assignment operator for `array` require that `T` be `MoveConstructible` or `MoveAssignable`, respectively.

23.581.0.1 `array` specialized algorithms [array.special]

```
template <class T, size_t N> void swap(array<T,N>& x, array<T,N>& y) noexcept(noexcept(x.swap(y)));
```

1 *Effects:*

```
x.swap(y);
```

2 *Complexity:* Linear in `N`.

23.581.0.2 `array::size` [array.size]

```
template <class T, size_t N> constexpr size_type array<T,N>::size() const noexcept;
```

1 *Returns:* `N`

23.581.0.3 `array::data` [array.data]

```
T* data() noexcept;
const T* data() const noexcept;
```

1 *Returns:* `elems`.

23.581.0.4 `array::fill` [array.fill]

```
void fill(const T& u);
```

1 *Effects:* `fill_n(begin(), N, u)`

23.581.0.5 `array::swap` [array.swap]

```
void swap(array& y) noexcept(noexcept(swap(declval<T&>(), declval<T&>())));
```

1 *Effects:* `swap_ranges(begin(), end(), y.begin())`

2 *Throws:* Nothing unless one of the element-wise swap calls throws an exception.

3 *Note:* Unlike the `swap` function for other containers, `array::swap` takes linear time, may exit via an exception, and does not cause iterators to become associated with the other container.

23.581.0.6 Zero sized arrays [array.zero]

1 `array` shall provide support for the special case `N == 0`.

1 In the case that `N == 0`, `begin() == end() ==` unique value. The return value of `data()` is unspecified.

1

The effect of calling `front()` or `back()` for a zero-sized array is undefined.

1

Member function `swap()` shall have a *noexcept-specification* which is equivalent to `noexcept(true)`.

23.586.0.1 Tuple interface to class template `array` [array.tuple]

```
template <class T, size_t N>
struct tuple_size<array<T, N>>
  : integral_constant<size_t, N> { };
```

```
tuple_element<I, array<T, N> >::type
```

1 *Requires:* `I < N`. The program is ill-formed if `I` is out of bounds.

2

Value: The type T.

```
template <size_t I, class T, size_t N>
constexpr T& get(array<T, N>& a) noexcept;
```

1 *Requires:* `I < N`. The program is ill-formed if `I` is out of bounds.

2 *Returns:* A reference to the `I`th element of `a`, where indexing is zero-based.

```
template <size_t I, class T, size_t N>
constexpr T&& get(array<T, N>&& a) noexcept;
```

3 *Effects:* Equivalent to `return std::move(get<I>(a));`

```
template <size_t I, class T, size_t N>
constexpr const T& get(const array<T, N>& a) noexcept;
```

4 *Requires:* `I < N`. The program is ill-formed if `I` is out of bounds.

5 *Returns:* A const reference to the `I`th element of `a`, where indexing is zero-based.

23.588.1 Class template deque [deque]

23.588.1.1 Class template deque overview [deque.overview]

1

A deque is a sequence container that, like a vector (23.696.1), supports random access iterators. In addition, it supports constant time insert and erase operations at the beginning or the end; insert and erase in the middle take linear time. That is, a deque is especially optimized for pushing and popping elements at the beginning and end. As with vectors, storage management is handled automatically.

1 A deque satisfies all of the requirements of a container, of a reversible container (given in tables in 23.2), of a sequence container, including the optional sequence container requirements (23.224.1), and of an allocator-aware container (Table 99). Descriptions are provided here only for operations on deque that are not described in one of these tables or for operations where there is additional semantic information.

```
namespace std {
  template <class T, class Allocator = allocator<T> >
  class deque {
  public:
    // types:
    typedef value_type&                          reference;
    typedef const value_type&                    const_reference;
    typedef implementation-defined               iterator;       // See 23.2
    typedef implementation-defined               const_iterator; // See 23.2
    typedef implementation-defined               size_type;      // See 23.2
    typedef implementation-defined               difference_type;// See 23.2
    typedef T                                    value_type;
    typedef Allocator                            allocator_type;
    typedef typename allocator_traits<Allocator>::pointer          pointer;
    typedef typename allocator_traits<Allocator>::const_pointer    const_pointer;
    typedef std::reverse_iterator<iterator>      reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

    // 23.592.0.1, construct/copy/destroy:
    deque() : deque(Allocator()) { }
    explicit deque(const Allocator&);
    explicit deque(size_type n, const Allocator& = Allocator());
    deque(size_type n, const T& value, const Allocator& = Allocator());
    template <class InputIterator>
      deque(InputIterator first, InputIterator last, const Allocator& = Allocator());
    deque(const deque& x);
    deque(deque&&);
    deque(const deque&, const Allocator&);
    deque(deque&&, const Allocator&);
    deque(initializer_list<T>, const Allocator& = Allocator());

    ~deque();
    deque& operator=(const deque& x);
    deque& operator=(deque&& x);
    deque& operator=(initializer_list<T>);
    template <class InputIterator>
      void assign(InputIterator first, InputIterator last);
    void assign(size_type n, const T& t);
    void assign(initializer_list<T>);
    allocator_type get_allocator() const noexcept;
```

```
// iterators:
iterator               begin() noexcept;
const_iterator         begin() const noexcept;
iterator               end() noexcept;
const_iterator         end() const noexcept;
reverse_iterator       rbegin() noexcept;
const_reverse_iterator rbegin() const noexcept;
reverse_iterator       rend() noexcept;
const_reverse_iterator rend() const noexcept;

const_iterator         cbegin() const noexcept;
const_iterator         cend() const noexcept;
const_reverse_iterator crbegin() const noexcept;
const_reverse_iterator crend() const noexcept;

// 23.597.0.1, capacity:
size_type size() const noexcept;
size_type max_size() const noexcept;
void      resize(size_type sz);
void      resize(size_type sz, const T& c);
void      shrink_to_fit();
bool      empty() const noexcept;

// element access:
reference       operator[](size_type n);
const_reference operator[](size_type n) const;
reference       at(size_type n);
const_reference at(size_type n) const;
reference       front();
const_reference front() const;
reference       back();
const_reference back() const;

// 23.610.0.1, modifiers:
template <class... Args> void emplace_front(Args&&... args);
template <class... Args> void emplace_back(Args&&... args);
template <class... Args> iterator emplace(const_iterator position, Args&&... args);

void push_front(const T& x);
void push_front(T&& x);
void push_back(const T& x);
void push_back(T&& x);

iterator insert(const_iterator position, const T& x);
iterator insert(const_iterator position, T&& x);
iterator insert(const_iterator position, size_type n, const T& x);
template <class InputIterator>
  iterator insert (const_iterator position, InputIterator first, InputIterator last);
iterator insert(const_iterator position, initializer_list<T>);

void pop_front();
void pop_back();

iterator erase(const_iterator position);
iterator erase(const_iterator first, const_iterator last);
```

```
    void     swap(deque&);
    void     clear() noexcept;
  };

  template <class T, class Allocator>
    bool operator==(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator< (const deque<T,Allocator>& x, const deque<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator!=(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator> (const deque<T,Allocator>& x, const deque<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator>=(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator<=(const deque<T,Allocator>& x, const deque<T,Allocator>& y);

  // specialized algorithms:
  template <class T, class Allocator>
    void swap(deque<T,Allocator>& x, deque<T,Allocator>& y);
}
```

23.592.0.1 `deque` constructors, copy, and assignment [deque.cons]

`explicit deque(const Allocator&);`

1

Effects: Constructs an empty `deque`, using the specified allocator.

1 *Complexity:* Constant.

`explicit deque(size_type n, const Allocator& = Allocator());`

2 *Effects:* Constructs a `deque` with `n` default-inserted elements using the specified allocator.

3 *Requires:* `T` shall be `DefaultInsertable` into `*this`.

4 *Complexity:* Linear in `n`.

```
deque(size_type n, const T& value,
      const Allocator& = Allocator());
```

5

Effects: Constructs a `deque` with `n` copies of `value`, using the specified allocator.

1 *Requires:* `T` shall be `CopyInsertable` into `*this`.

2

Complexity: Linear in `n`.

```
template <class InputIterator>
  deque(InputIterator first, InputIterator last,
        const Allocator& = Allocator());
```

1 *Effects:* Constructs a `deque` equal to the range `[first,last)`, using the specified allocator.

2 *Complexity:* Linear in `distance(first, last)`.

23.597.0.1 deque capacity [deque.capacity]

```
void resize(size_type sz);
```

1

Effects: If `sz <= size()`, equivalent to calling `pop_back() size() - sz` times. If `size() < sz`, appends `sz - size()` default-inserted elements to the sequence.

1 *Requires:* `T` shall be `MoveInsertable` and `DefaultInsertable` into `*this`.

```
void resize(size_type sz, const T& c);
```

2 *Effects:* If `sz <= size()`, equivalent to calling `pop_back() size() - sz` times. If `size() < sz`, appends `sz - size()` copies of `c` to the sequence.

3 *Requires:* `T` shall be `CopyInsertable` into `*this`.

```
void shrink_to_fit();
```

4 *Requires:* `T` shall be `MoveInsertable` into `*this`.

5 *Complexity:* Linear in the size of the sequence.

6 *Remarks:* `shrink_to_fit` is a non-binding request to reduce memory use but does not change the size of the sequence. [*Note:* The request is non-binding to allow latitude for implementation-specific optimizations. — *end note*]

23.610.0.1 deque modifiers [deque.modifiers]

```
iterator insert(const_iterator position, const T& x);
iterator insert(const_iterator position, T&& x);
iterator insert(const_iterator position, size_type n, const T& x);
template <class InputIterator>
  iterator insert(const_iterator position,
                  InputIterator first, InputIterator last);
iterator insert(const_iterator position, initializer_list<T>);

template <class... Args> void emplace_front(Args&&... args);
template <class... Args> void emplace_back(Args&&... args);
template <class... Args> iterator emplace(const_iterator position, Args&&... args);
void push_front(const T& x);
void push_front(T&& x);
void push_back(const T& x);
void push_back(T&& x);
```

1

Effects: An insertion in the middle of the deque invalidates all the iterators and references to elements of the deque. An insertion at either end of the deque invalidates all the iterators to the deque, but has no effect on the validity of references to elements of the deque.

1

Remarks: If an exception is thrown other than by the copy constructor, move constructor, assignment operator, or move assignment operator of `T` there are no effects. If an exception is thrown while inserting a single element at either end, there are no effects. Otherwise, if an exception is thrown by the move constructor of a non-`CopyInsertable` `T`, the effects are unspecified.

1

Complexity: The complexity is linear in the number of elements inserted plus the lesser of the distances to the beginning and end of the deque. Inserting a single element either at the beginning or end of a deque always takes constant time and causes a single call to a constructor of `T`.

```
iterator erase(const_iterator position);
iterator erase(const_iterator first, const_iterator last);
```

1

Effects: An erase operation that erases the last element of a deque invalidates only the past-the-end iterator and all iterators and references to the erased elements. An erase operation that erases the first element of a deque but not the last element invalidates only the erased elements. An erase operation that erases neither the first element nor the last element of a deque invalidates the past-the-end iterator and all iterators and references to all the elements of the deque.

1

Complexity: The number of calls to the destructor is the same as the number of elements erased, but the number of calls to the assignment operator is no more than the lesser of the number of elements before the erased elements and the number of elements after the erased elements.

1

Throws: Nothing unless an exception is thrown by the copy constructor, move constructor, assignment operator, or move assignment operator of `T`.

23.652.0.1 `deque` specialized algorithms [deque.special]

```
template <class T, class Allocator>
  void swap(deque<T,Allocator>& x, deque<T,Allocator>& y);
```

1 *Effects:*

```
x.swap(y);
```

23.652.1 Class template `forward_list` [forwardlist]

23.652.1.1 Class template `forward_list` overview [forwardlist.overview]

1

A `forward_list` is a container that supports forward iterators and allows constant time insert and erase operations anywhere within the sequence, with storage management handled automatically. Fast random access to list elements is not supported. [*Note:* It is intended that `forward_list` have zero space or time overhead relative to a hand-written C-style singly linked list. Features that would conflict with that goal have been omitted. — *end note*]

1

A `forward_list` satisfies all of the requirements of a container (Table 96), except that the `size()` member function is not provided and `operator==` has linear complexity. A `forward_list` also satisfies all of the requirements for an allocator-aware container (Table 99). In addition, a `forward_list` provides the `assign` member functions (Table 100) and several of the optional container requirements (Table 101). Descriptions are provided here only for operations on `forward_list` that are not described in that table or for operations where there is additional semantic information.

1 [*Note:* Modifying any list requires access to the element preceding the first element of interest, but in a `forward_list` there is no constant-time way to access a preceding element. For this reason, ranges that are modified, such as those supplied to `erase` and `splice`, must be open at the beginning. — *end note*]

```
namespace std {
  template <class T, class Allocator = allocator<T> >
  class forward_list {
  public:
    // types:
```

```
typedef value_type&                                          reference;
typedef const value_type&                                    const_reference;
typedef implementation-defined iterator;       // See 23.2
typedef implementation-defined const_iterator; // See 23.2
typedef implementation-defined size_type;      // See 23.2
typedef implementation-defined difference_type;// See 23.2
typedef T value_type;
typedef Allocator allocator_type;
typedef typename allocator_traits<Allocator>::pointer         pointer;
typedef typename allocator_traits<Allocator>::const_pointer   const_pointer;

// 23.655.0.1, construct/copy/destroy:
forward_list() : forward_list(Allocator()) { }
explicit forward_list(const Allocator&);
explicit forward_list(size_type n, const Allocator& = Allocator());
forward_list(size_type n, const T& value,
             const Allocator& = Allocator());
template <class InputIterator>
  forward_list(InputIterator first, InputIterator last,
               const Allocator& = Allocator());
forward_list(const forward_list& x);
forward_list(forward_list&& x);
forward_list(const forward_list& x, const Allocator&);
forward_list(forward_list&& x, const Allocator&);
forward_list(initializer_list<T>, const Allocator& = Allocator());
~forward_list();
forward_list& operator=(const forward_list& x);
forward_list& operator=(forward_list&& x);
forward_list& operator=(initializer_list<T>);
template <class InputIterator>
  void assign(InputIterator first, InputIterator last);
void assign(size_type n, const T& t);
void assign(initializer_list<T>);
allocator_type get_allocator() const noexcept;

// 23.655.0.2, iterators:
iterator before_begin() noexcept;
const_iterator before_begin() const noexcept;
iterator begin() noexcept;
const_iterator begin() const noexcept;
iterator end() noexcept;
const_iterator end() const noexcept;

const_iterator cbegin() const noexcept;
const_iterator cbefore_begin() const noexcept;
const_iterator cend() const noexcept;

// capacity:
bool empty() const noexcept;
size_type max_size() const noexcept;

// 23.656.0.1, element access:
reference front();
const_reference front() const;
```

```
  // 23.656.0.2, modifiers:
  template <class... Args> void emplace_front(Args&&... args);
  void push_front(const T& x);
  void push_front(T&& x);
  void pop_front();

  template <class... Args> iterator emplace_after(const_iterator position, Args&&... args);
  iterator insert_after(const_iterator position, const T& x);
  iterator insert_after(const_iterator position, T&& x);

  iterator insert_after(const_iterator position, size_type n, const T& x);
  template <class InputIterator>
    iterator insert_after(const_iterator position, InputIterator first, InputIterator last);
  iterator insert_after(const_iterator position, initializer_list<T> il);

  iterator erase_after(const_iterator position);
  iterator erase_after(const_iterator position, const_iterator last);
  void swap(forward_list&);

  void resize(size_type sz);
  void resize(size_type sz, const value_type& c);
  void clear() noexcept;

  // 23.662.0.1, forward_list operations:
  void splice_after(const_iterator position, forward_list& x);
  void splice_after(const_iterator position, forward_list&& x);
  void splice_after(const_iterator position, forward_list& x,
                    const_iterator i);
  void splice_after(const_iterator position, forward_list&& x,
                    const_iterator i);
  void splice_after(const_iterator position, forward_list& x,
                    const_iterator first, const_iterator last);
  void splice_after(const_iterator position, forward_list&& x,
                    const_iterator first, const_iterator last);

  void remove(const T& value);
  template <class Predicate> void remove_if(Predicate pred);

  void unique();
  template <class BinaryPredicate> void unique(BinaryPredicate binary_pred);

  void merge(forward_list& x);
  void merge(forward_list&& x);
  template <class Compare> void merge(forward_list& x, Compare comp);
  template <class Compare> void merge(forward_list&& x, Compare comp);

  void sort();
  template <class Compare> void sort(Compare comp);

  void reverse() noexcept;
};

// Comparison operators
template <class T, class Allocator>
  bool operator==(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
```

```
  template <class T, class Allocator>
    bool operator< (const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator!=(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator> (const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator>=(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator<=(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);

  // 23.672.0.1, specialized algorithms:
  template <class T, class Allocator>
    void swap(forward_list<T,Allocator>& x, forward_list<T,Allocator>& y);
}
```

23.655.0.1 `forward_list` constructors, copy, assignment [forwardlist.cons]

`explicit forward_list(const Allocator&);`

1 *Effects:* Constructs an empty `forward_list` object using the specified allocator.

2 *Complexity:* Constant.

`explicit forward_list(size_type n, const Allocator& = Allocator());`

3 *Effects:* Constructs a `forward_list` object with `n` default-inserted elements using the specified allocator.

4 *Requires:* `T` shall be `DefaultInsertable` into `*this`.

5 *Complexity:* Linear in `n`.

`forward_list(size_type n, const T& value, const Allocator& = Allocator());`

6 *Effects:* Constructs a `forward_list` object with `n` copies of `value` using the specified allocator.

7 *Requires:* `T` shall be `CopyInsertable` into `*this`.

8 *Complexity:* Linear in `n`.

```
template <class InputIterator>
  forward_list(InputIterator first, InputIterator last, const Allocator& = Allocator());
```

9 *Effects:* Constructs a `forward_list` object equal to the range `[first,last)`.

10 *Complexity:* Linear in `distance(first, last)`.

23.655.0.2 `forward_list` iterators [forwardlist.iter]

```
iterator before_begin() noexcept;
const_iterator before_begin() const noexcept;
const_iterator cbefore_begin() const noexcept;
```

1

Returns: A non-dereferenceable iterator that, when incremented, is equal to the iterator returned by `begin()`.

1 *Effects:* `cbefore_begin()` is equivalent to `const_cast<forward_list const&>(*this).before_begin()`.

2 *Remarks:* `before_begin() == end()` shall equal `false`.

23.656.0.1 `forward_list` element access [forwardlist.access]

```
reference front();
const_reference front() const;
```

1 *Returns:* `*begin()`

23.656.0.2 `forward_list` modifiers [forwardlist.modifiers]

1

None of the overloads of `insert_after` shall affect the validity of iterators and references, and `erase_after` shall invalidate only iterators and references to the erased elements. If an exception is thrown during `insert_after` there shall be no effect. Inserting `n` elements into a `forward_list` is linear in `n`, and the number of calls to the copy or move constructor of `T` is exactly equal to `n`. Erasing `n` elements from a `forward_list` is linear in `n` and the number of calls to the destructor of type `T` is exactly equal to `n`.

```
template <class... Args> void emplace_front(Args&&... args);
```

1 *Effects:* Inserts an object of type `value_type` constructed with `value_type(std::forward<Args>(args)...)` at the beginning of the list.

```
void push_front(const T& x);
void push_front(T&& x);
```

2 *Effects:* Inserts a copy of `x` at the beginning of the list.

```
void pop_front();
```

3 *Effects:* `erase_after(before_begin())`

```
iterator insert_after(const_iterator position, const T& x);
iterator insert_after(const_iterator position, T&& x);
```

4 *Requires:* `position` is `before_begin()` or is a dereferenceable iterator in the range `[begin(),end())`.

5 *Effects:* Inserts a copy of `x` after `position`.

6

Returns: An iterator pointing to the copy of `x`.

```
iterator insert_after(const_iterator position, size_type n, const T& x);
```

1 *Requires:* `position` is `before_begin()` or is a dereferenceable iterator in the range `[begin(),end())`.

2 *Effects:* Inserts `n` copies of `x` after `position`.

3

Returns: An iterator pointing to the last inserted copy of `x` or `position` if `n == 0`.

```
template <class InputIterator>
  iterator insert_after(const_iterator position, InputIterator first, InputIterator last);
```

1 *Requires:* `position` is `before_begin()` or is a dereferenceable iterator in the range `[begin(),end())`. `first` and `last` are not iterators in `*this`.

2 *Effects:* Inserts copies of elements in `[first,last)` after `position`.

3 *Returns:* An iterator pointing to the last inserted element or `position` if `first == last`.

```
iterator insert_after(const_iterator position, initializer_list<T> il);
```

4 *Effects:* `insert_after(p, il.begin(), il.end())`.

5 *Returns:* An iterator pointing to the last inserted element or `position` if `il` is empty.

```
template <class... Args>
  iterator emplace_after(const_iterator position, Args&&... args);
```

1 *Requires:* `position` is `before_begin()` or is a dereferenceable iterator in the range `[begin(),end())`.

2 *Effects:* Inserts an object of type `value_type` constructed with `value_type(std::forward<Args>(args)...)` after `position`.

3 *Returns:* An iterator pointing to the new object.

```
iterator erase_after(const_iterator position);
```

4 *Requires:* The iterator following `position` is dereferenceable.

5 *Effects:* Erases the element pointed to by the iterator following `position`.

6 *Returns:* An iterator pointing to the element following the one that was erased, or `end()` if no such element exists.

7 *Throws:* Nothing.

```
iterator erase_after(const_iterator position, const_iterator last);
```

8 *Requires:* All iterators in the range `(position,last)` are dereferenceable.

9 *Effects:* Erases the elements in the range `(position,last)`.

10 *Returns:* `last`.

11 *Throws:* Nothing.

```
void resize(size_type sz);
```

12 *Effects:* If `sz < distance(begin(), end())`, erases the last `distance(begin(), end()) - sz` elements from the list. Otherwise, inserts `sz - distance(begin(), end())` default-inserted elements at the end of the list.

13 *Requires:* `T` shall be `DefaultInsertable` into `*this`.

```
void resize(size_type sz, const value_type& c);
```

14 *Effects:* If `sz < distance(begin(), end())`, erases the last `distance(begin(), end()) - sz` elements from the list. Otherwise, inserts `sz - distance(begin(), end())` elements at the end of the list such that each new element, `e`, is initialized by a method equivalent to calling `allocator_traits<allocator_type>::construct(get_allocator(), std::addressof(e), c)`.

1 *Requires:* `T` shall be `CopyInsertable` into `*this`.

```
void clear() noexcept;
```

2 *Effects:* Erases all elements in the range `[begin(),end())`.

3 *Remarks:* Does not invalidate past-the-end iterators.

23.662.0.1 `forward_list` operations [forwardlist.ops]

```
void splice_after(const_iterator position, forward_list& x);
void splice_after(const_iterator position, forward_list&& x);
```

1 *Requires:* `position` is `before_begin()` or is a dereferenceable iterator in the range `[begin(),end())`. `get_allocator() == x.get_allocator()`. `&x != this`.

2 *Effects:* Inserts the contents of `x` after `position`, and `x` becomes empty. Pointers and references to the moved elements of `x` now refer to those same elements but as members of `*this`. Iterators referring to the moved elements will continue to refer to their elements, but they now behave as iterators into `*this`, not into `x`.

1 *Throws:* Nothing.

2 *Complexity:* $\mathcal{O}(distance(x.begin(), x.end()))$

```
void splice_after(const_iterator position, forward_list& x, const_iterator i);
void splice_after(const_iterator position, forward_list&& x, const_iterator i);
```

3 *Requires:* `position` is `before_begin()` or is a dereferenceable iterator in the range `[begin(),end())`. The iterator following `i` is a dereferenceable iterator in `x`. `get_allocator() == x.get_allocator()`.

4 *Effects:* Inserts the element following `i` into `*this`, following `position`, and removes it from `x`. The result is unchanged if `position == i` or `position == ++i`. Pointers and references to `*++i` continue to refer to the same element but as a member of `*this`. Iterators to `*++i` continue to refer to the same element, but now behave as iterators into `*this`, not into `x`.

5 *Throws:* Nothing.

6 *Complexity:* $\mathcal{O}(1)$

```
void splice_after(const_iterator position, forward_list& x,
                  const_iterator first, const_iterator last);
void splice_after(const_iterator position, forward_list&& x,
                  const_iterator first, const_iterator last);
```

7 *Requires:* `position` is `before_begin()` or is a dereferenceable iterator in the range `[begin(),end())`. `(first,last)` is a valid range in `x`, and all iterators in the range `(first,last)` are dereferenceable. `position` is not an iterator in the range `(first,last)`. `get_allocator() == x.get_allocator()`.

8 *Effects:* Inserts elements in the range `(first,last)` after `position` and removes the elements from `x`. Pointers and references to the moved elements of `x` now refer to those same elements but as members of `*this`. Iterators referring to the moved elements will continue to refer to their elements, but they now behave as iterators into `*this`, not into `x`.

9 *Complexity:* $\mathcal{O}(distance(first, last))$

```
void remove(const T& value);
template <class Predicate> void remove_if(Predicate pred);
```

10 *Effects:* Erases all the elements in the list referred by a list iterator `i` for which the following conditions hold: `*i == value` (for `remove()`), `pred(*i)` is true (for `remove_if()`). Invalidates only the iterators and references to the erased elements.

1 *Throws:* Nothing unless an exception is thrown by the equality comparison or the predicate.

1 *Remarks:* Stable (17.602.0.1).

2

Complexity: Exactly `distance(begin(), end())` applications of the corresponding predicate.

```
void unique();
template <class BinaryPredicate> void unique(BinaryPredicate pred);
```

1 *Effects:* Erases all but the first element from every consecutive group of equal elements referred to by the iterator `i` in the range `[first + 1,last)` for which `*i == *(i-1)` (for the version with no arguments) or `pred(*i, *(i - 1))` (for the version with a predicate argument) holds. Invalidates only the iterators and references to the erased elements.

2 *Throws:* Nothing unless an exception is thrown by the equality comparison or the predicate.

3 *Complexity:* If the range `[first,last)` is not empty, exactly `(last - first) - 1` applications of the corresponding predicate, otherwise no applications of the predicate.

```
void merge(forward_list& x);
void merge(forward_list&& x);
template <class Compare> void merge(forward_list& x, Compare comp);
template <class Compare> void merge(forward_list&& x, Compare comp);
```

4 *Requires:* `comp` defines a strict weak ordering (25.98), and `*this` and `x` are both sorted according to this ordering. `get_allocator() == x.get_allocator()`.

5 *Effects:* Merges the two sorted ranges `[begin(), end())` and `[x.begin(), x.end())`. `x` is empty after the merge. If an exception is thrown other than by a comparison there are no effects. Pointers and references to the moved elements of `x` now refer to those same elements but as members of `*this`. Iterators referring to the moved elements will continue to refer to their elements, but they now behave as iterators into `*this`, not into `x`.

6 *Remarks:* Stable (17.602.0.1). The behavior is undefined if `this->get_allocator() != x.get_allocator()`.

7 *Complexity:* At most `distance(begin(), end()) + distance(x.begin(), x.end()) - 1` comparisons.

```
void sort();
template <class Compare> void sort(Compare comp);
```

8 *Requires:* `operator<` (for the version with no arguments) or `comp` (for the version with a comparison argument) defines a strict weak ordering (25.98).

9

Effects: Sorts the list according to the `operator<` or the `comp` function object. If an exception is thrown the order of the elements in `*this` is unspecified. Does not affect the validity of iterators and references.

1 *Remarks:* Stable (17.602.0.1).

2

Complexity: Approximately $N \log N$ comparisons, where N is `distance(begin(), end())`.

```
void reverse() noexcept;
```

1 *Effects:* Reverses the order of the elements in the list. Does not affect the validity of iterators and references.

2 *Complexity:* Linear time.

23.672.0.1 `forward_list` specialized algorithms [forwardlist.spec]

```
template <class T, class Allocator>
  void swap(forward_list<T,Allocator>& x, forward_list<T,Allocator>& y);
```

1 *Effects:* `x.swap(y)`

23.672.1 Class template `list` [list]

23.672.1.1 Class template `list` overview [list.overview]

1

A `list` is a sequence container that supports bidirectional iterators and allows constant time insert and erase operations anywhere within the sequence, with storage management handled automatically. Unlike vectors (23.696.1) and deques (23.588.1), fast random access to list elements is not supported, but many algorithms only need sequential access anyway.

1

A `list` satisfies all of the requirements of a container, of a reversible container (given in two tables in 23.2), of a sequence container, including most of the optional sequence container requirements (23.224.1), and of an allocator-aware container (Table 99). The exceptions are the `operator[]` and `at` member functions, which are not provided.[264] Descriptions are provided here only for operations on `list` that are not described in one of these tables or for operations where there is additional semantic information.

```
namespace std {
  template <class T, class Allocator = allocator<T> >
  class list {
  public:
    // types:
    typedef value_type&                                                reference;
    typedef const value_type&                                          const_reference;
    typedef implementation-defined                iterator;       // see 23.2
    typedef implementation-defined                const_iterator; // see 23.2
    typedef implementation-defined                size_type;      // see 23.2
    typedef implementation-defined                difference_type;// see 23.2
    typedef T                                     value_type;
    typedef Allocator                             allocator_type;
    typedef typename allocator_traits<Allocator>::pointer           pointer;
    typedef typename allocator_traits<Allocator>::const_pointer     const_pointer;
    typedef std::reverse_iterator<iterator>       reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

    // 23.678.0.1, construct/copy/destroy:
    list() : list(Allocator()) { }
    explicit list(const Allocator&);
    explicit list(size_type n, const Allocator& = Allocator());
    list(size_type n, const T& value, const Allocator& = Allocator());
    template <class InputIterator>
      list(InputIterator first, InputIterator last, const Allocator& = Allocator());
    list(const list& x);
    list(list&& x);
    list(const list&, const Allocator&);
    list(list&&, const Allocator&);
    list(initializer_list<T>, const Allocator& = Allocator());
   ~list();
```

264) These member functions are only provided by containers whose iterators are random access iterators.

```
list& operator=(const list& x);
list& operator=(list&& x);
list& operator=(initializer_list<T>);
template <class InputIterator>
  void assign(InputIterator first, InputIterator last);
void assign(size_type n, const T& t);
void assign(initializer_list<T>);
allocator_type get_allocator() const noexcept;

// iterators:
iterator               begin() noexcept;
const_iterator         begin() const noexcept;
iterator               end() noexcept;
const_iterator         end() const noexcept;
reverse_iterator       rbegin() noexcept;
const_reverse_iterator rbegin() const noexcept;
reverse_iterator       rend() noexcept;
const_reverse_iterator rend() const noexcept;

const_iterator         cbegin() const noexcept;
const_iterator         cend() const noexcept;
const_reverse_iterator crbegin() const noexcept;
const_reverse_iterator crend() const noexcept;

// 23.678.0.2, capacity:
bool      empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;
void      resize(size_type sz);
void      resize(size_type sz, const T& c);

// element access:
reference       front();
const_reference front() const;
reference       back();
const_reference back() const;

// 23.681.0.1, modifiers:
template <class... Args> void emplace_front(Args&&... args);
void pop_front();
template <class... Args> void emplace_back(Args&&... args);
void push_front(const T& x);
void push_front(T&& x);
void push_back(const T& x);
void push_back(T&& x);
void pop_back();

template <class... Args> iterator emplace(const_iterator position, Args&&... args);
iterator insert(const_iterator position, const T& x);
iterator insert(const_iterator position, T&& x);
iterator insert(const_iterator position, size_type n, const T& x);
template <class InputIterator>
  iterator insert(const_iterator position, InputIterator first,
                  InputIterator last);
iterator insert(const_iterator position, initializer_list<T> il);
```

```
    iterator erase(const_iterator position);
    iterator erase(const_iterator position, const_iterator last);
    void     swap(list&);
    void     clear() noexcept;

    // 23.682.0.1, list operations:
    void splice(const_iterator position, list& x);
    void splice(const_iterator position, list&& x);
    void splice(const_iterator position, list& x, const_iterator i);
    void splice(const_iterator position, list&& x, const_iterator i);
    void splice(const_iterator position, list& x,
                const_iterator first, const_iterator last);
    void splice(const_iterator position, list&& x,
                const_iterator first, const_iterator last);

    void remove(const T& value);
    template <class Predicate> void remove_if(Predicate pred);

    void unique();
    template <class BinaryPredicate>
      void unique(BinaryPredicate binary_pred);

    void merge(list& x);
    void merge(list&& x);
    template <class Compare> void merge(list& x, Compare comp);
    template <class Compare> void merge(list&& x, Compare comp);

    void sort();
    template <class Compare> void sort(Compare comp);

    void reverse() noexcept;
  };

  template <class T, class Allocator>
    bool operator==(const list<T,Allocator>& x, const list<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator< (const list<T,Allocator>& x, const list<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator!=(const list<T,Allocator>& x, const list<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator> (const list<T,Allocator>& x, const list<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator>=(const list<T,Allocator>& x, const list<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator<=(const list<T,Allocator>& x, const list<T,Allocator>& y);

  // specialized algorithms:
  template <class T, class Allocator>
    void swap(list<T,Allocator>& x, list<T,Allocator>& y);
}
```

23.678.0.1 `list` constructors, copy, and assignment [list.cons]

```
explicit list(const Allocator&);
```

1 *Effects:* Constructs an empty list, using the specified allocator.

2 *Complexity:* Constant.

```
explicit list(size_type n, const Allocator& = Allocator());
```

3 *Effects:* Constructs a `list` with `n` default-inserted elements using the specified allocator.

4 *Requires:* `T` shall be `DefaultInsertable` into `*this`.

5 *Complexity:* Linear in `n`.

```
list(size_type n, const T& value,
     const Allocator& = Allocator());
```

6 *Effects:* Constructs a `list` with `n` copies of `value`, using the specified allocator.

7 *Requires:* `T` shall be `CopyInsertable` into `*this`.

8 *Complexity:* Linear in `n`.

```
template <class InputIterator>
list(InputIterator first, InputIterator last,
     const Allocator& = Allocator());
```

9 *Effects:* Constructs a `list` equal to the range `[first,last)`.

10 *Complexity:* Linear in `distance(first, last)`.

23.678.0.2 `list` capacity [list.capacity]

```
void resize(size_type sz);
```

1

Effects: If `size() < sz`, appends `sz - size()` default-inserted elements to the sequence. If `sz <= size()`, equivalent to

```
list<T>::iterator it = begin();
advance(it, sz);
erase(it, end());
```

1 *Requires:* `T` shall be `DefaultInsertable` into `*this`.

```
void resize(size_type sz, const T& c);
```

2 *Effects:*

```
if (sz > size())
  insert(end(), sz-size(), c);
else if (sz < size()) {
  iterator i = begin();
  advance(i, sz);
  erase(i, end());
}
else
  ;                    // do nothing
```

3 *Requires:* `T` shall be `CopyInsertable` into `*this`.

23.681.0.1 `list` modifiers [list.modifiers]

```
iterator insert(const_iterator position, const T& x);
iterator insert(const_iterator position, T&& x);
iterator insert(const_iterator position, size_type n, const T& x);
template <class InputIterator>
  iterator insert(const_iterator position, InputIterator first,
                  InputIterator last);
iterator insert(const_iterator position, initializer_list<T>);

template <class... Args> void emplace_front(Args&&... args);
template <class... Args> void emplace_back(Args&&... args);
template <class... Args> iterator emplace(const_iterator position, Args&&... args);
void push_front(const T& x);
void push_front(T&& x);
void push_back(const T& x);
void push_back(T&& x);
```

1 *Remarks:* Does not affect the validity of iterators and references. If an exception is thrown there are no effects.

2

Complexity: Insertion of a single element into a list takes constant time and exactly one call to a constructor of `T`. Insertion of multiple elements into a list is linear in the number of elements inserted, and the number of calls to the copy constructor or move constructor of `T` is exactly equal to the number of elements inserted.

```
iterator erase(const_iterator position);
iterator erase(const_iterator first, const_iterator last);

void pop_front();
void pop_back();
void clear() noexcept;
```

1 *Effects:* Invalidates only the iterators and references to the erased elements.

2 *Throws:* Nothing.

3 *Complexity:* Erasing a single element is a constant time operation with a single call to the destructor of `T`. Erasing a range in a list is linear time in the size of the range and the number of calls to the destructor of type `T` is exactly equal to the size of the range.

23.682.0.1 `list` operations [list.ops]

1 Since lists allow fast insertion and erasing from the middle of a list, certain operations are provided specifically for them.[265]

2

`list` provides three splice operations that destructively move elements from one list to another. The behavior of splice operations is undefined if `get_allocator() != x.get_allocator()`.

```
void splice(const_iterator position, list& x);
void splice(const_iterator position, list&& x);
```

1 *Requires:* `&x != this`.

2

265) As specified in 17.165.0.1, the requirements in this Clause apply only to lists whose allocators compare equal.

Effects: Inserts the contents of `x` before `position` and `x` becomes empty. Pointers and references to the moved elements of `x` now refer to those same elements but as members of `*this`. Iterators referring to the moved elements will continue to refer to their elements, but they now behave as iterators into `*this`, not into `x`.

1 *Throws:* Nothing.

2 *Complexity:* Constant time.

```
void splice(const_iterator position, list& x, const_iterator i);
void splice(const_iterator position, list&& x, const_iterator i);
```

1 *Effects:* Inserts an element pointed to by `i` from list `x` before `position` and removes the element from `x`. The result is unchanged if `position == i` or `position == ++i`. Pointers and references to `*i` continue to refer to this same element but as a member of `*this`. Iterators to `*i` (including `i` itself) continue to refer to the same element, but now behave as iterators into `*this`, not into `x`.

1 *Requires:* `i` is a valid dereferenceable iterator of `x`.

2 *Throws:* Nothing.

3 *Complexity:* Constant time.

```
void splice(const_iterator position, list& x, const_iterator first,
            const_iterator last);
void splice(const_iterator position, list&& x, const_iterator first,
            const_iterator last);
```

4 *Effects:* Inserts elements in the range `[first,last)` before `position` and removes the elements from `x`.

5 *Requires:* `[first, last)` is a valid range in `x`. The result is undefined if `position` is an iterator in the range `[first,last)`. Pointers and references to the moved elements of `x` now refer to those same elements but as members of `*this`. Iterators referring to the moved elements will continue to refer to their elements, but they now behave as iterators into `*this`, not into `x`.

1 *Throws:* Nothing.

2 *Complexity:* Constant time if `&x == this`; otherwise, linear time.

```
                           void remove(const T& value);
template <class Predicate> void remove_if(Predicate pred);
```

1 *Effects:* Erases all the elements in the list referred by a list iterator `i` for which the following conditions hold: `*i == value, pred(*i) != false`. Invalidates only the iterators and references to the erased elements.

1 *Throws:* Nothing unless an exception is thrown by `*i == value` or `pred(*i) != false`.

2 *Remarks:* Stable (17.602.0.1).

3 *Complexity:* Exactly `size()` applications of the corresponding predicate.

```
                                 void unique();
template <class BinaryPredicate> void unique(BinaryPredicate binary_pred);
```

4 *Effects:* Erases all but the first element from every consecutive group of equal elements referred to by the iterator `i` in the range `[first + 1,last)` for which `*i == *(i-1)` (for the version of `unique` with no arguments) or `pred(*i, *(i - 1))` (for the version of `unique` with a predicate argument) holds. Invalidates only the iterators and references to the erased elements.

5 *Throws:* Nothing unless an exception is thrown by `*i == *(i-1)` or `pred(*i, *(i - 1))`

6 *Complexity:* If the range `[first, last)` is not empty, exactly `(last - first) - 1` applications of the corresponding predicate, otherwise no applications of the predicate.

```
void                          merge(list& x);
void                          merge(list&& x);
template <class Compare> void merge(list& x, Compare comp);
template <class Compare> void merge(list&& x, Compare comp);
```

1 *Requires:* `comp` shall define a strict weak ordering (25.98), and both the list and the argument list shall be sorted according to this ordering.

1 *Effects:* If `(&x == this)` does nothing; otherwise, merges the two sorted ranges `[begin(), end())` and `[x.begin(), x.end())`. The result is a range in which the elements will be sorted in non-decreasing order according to the ordering defined by `comp`; that is, for every iterator `i`, in the range other than the first, the condition `comp(*i, *(i - 1))` will be false. Pointers and references to the moved elements of `x` now refer to those same elements but as members of `*this`. Iterators referring to the moved elements will continue to refer to their elements, but they now behave as iterators into `*this`, not into `x`.

2 *Remarks:* Stable (17.602.0.1). If `(&x != this)` the range `[x.begin(), x.end())` is empty after the merge. No elements are copied by this operation. The behavior is undefined if `this->get_allocator() != x.get_allocator()`.

3 *Complexity:* At most `size() + x.size() - 1` applications of `comp` if `(&x != this)`; otherwise, no applications of `comp` are performed. If an exception is thrown other than by a comparison there are no effects.

```
void reverse() noexcept;
```

4 *Effects:* Reverses the order of the elements in the list. Does not affect the validity of iterators and references.

5 *Complexity:* Linear time.

```
                          void sort();
template <class Compare> void sort(Compare comp);
```

6 *Requires:* `operator<` (for the first version) or `comp` (for the second version) shall define a strict weak ordering (25.98).

7 *Effects:* Sorts the list according to the `operator<` or a `Compare` function object. Does not affect the validity of iterators and references.

1 *Remarks:* Stable (17.602.0.1).

2 *Complexity:* Approximately $N \log(N)$ comparisons, where `N == size()`.

23.696.0.1 `list` specialized algorithms [list.special]

```
template <class T, class Allocator>
  void swap(list<T,Allocator>& x, list<T,Allocator>& y);
```

1 *Effects:*

```
x.swap(y);
```

23.696.1 Class template `vector` [vector]

23.696.1.1 Class template `vector` overview [vector.overview]

1

A `vector` is a sequence container that supports random access iterators. In addition, it supports (amortized) constant time insert and erase operations at the end; insert and erase in the middle take linear time. Storage management is handled automatically, though hints can be given to improve efficiency. The elements of a vector are stored contiguously, meaning that if `v` is a `vector<T, Allocator>` where `T` is some type other than `bool`, then it obeys the identity `&v[n] == &v[0] + n` for all `0 <= n < v.size()`.

1

A `vector` satisfies all of the requirements of a container and of a reversible container (given in two tables in 23.2), of a sequence container, including most of the optional sequence container requirements (23.224.1), and of an allocator-aware container (Table 99). The exceptions are the `push_front`, `pop_front`, and `emplace_front` member functions, which are not provided. Descriptions are provided here only for operations on `vector` that are not described in one of these tables or for operations where there is additional semantic information.

```
namespace std {
  template <class T, class Allocator = allocator<T> >
  class vector {
  public:
    // types:
    typedef value_type&                          reference;
    typedef const value_type&                    const_reference;
    typedef implementation-defined               iterator;       // see 23.2
    typedef implementation-defined               const_iterator; // see 23.2
    typedef implementation-defined               size_type;      // see 23.2
    typedef implementation-defined               difference_type;// see 23.2
    typedef T                                    value_type;
    typedef Allocator                            allocator_type;
    typedef typename allocator_traits<Allocator>::pointer           pointer;
    typedef typename allocator_traits<Allocator>::const_pointer     const_pointer;
    typedef std::reverse_iterator<iterator>      reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

    // 23.748.0.1, construct/copy/destroy:
    vector() : vector(Allocator()) { }
    explicit vector(const Allocator&);
    explicit vector(size_type n, const Allocator& = Allocator());
    vector(size_type n, const T& value, const Allocator& = Allocator());
    template <class InputIterator>
      vector(InputIterator first, InputIterator last,
             const Allocator& = Allocator());
    vector(const vector& x);
    vector(vector&&);
    vector(const vector&, const Allocator&);
```

```
 vector(vector&&, const Allocator&);
 vector(initializer_list<T>, const Allocator& = Allocator());
~vector();
 vector& operator=(const vector& x);
 vector& operator=(vector&& x);
 vector& operator=(initializer_list<T>);
 template <class InputIterator>
   void assign(InputIterator first, InputIterator last);
 void assign(size_type n, const T& u);
 void assign(initializer_list<T>);
 allocator_type get_allocator() const noexcept;

 // iterators:
 iterator               begin() noexcept;
 const_iterator         begin() const noexcept;
 iterator               end() noexcept;
 const_iterator         end() const noexcept;
 reverse_iterator       rbegin() noexcept;
 const_reverse_iterator rbegin() const noexcept;
 reverse_iterator       rend() noexcept;
 const_reverse_iterator rend() const noexcept;

 const_iterator         cbegin() const noexcept;
 const_iterator         cend() const noexcept;
 const_reverse_iterator crbegin() const noexcept;
 const_reverse_iterator crend() const noexcept;

 // 23.758.0.1, capacity:
 size_type size() const noexcept;
 size_type max_size() const noexcept;
 void      resize(size_type sz);
 void      resize(size_type sz, const T& c);
 size_type capacity() const noexcept;
 bool      empty() const noexcept;
 void      reserve(size_type n);
 void      shrink_to_fit();

 // element access:
 reference       operator[](size_type n);
 const_reference operator[](size_type n) const;
 const_reference at(size_type n) const;
 reference       at(size_type n);
 reference       front();
 const_reference front() const;
 reference       back();
 const_reference back() const;

 // 23.801.0.1, data access
 T*         data() noexcept;
 const T*  data() const noexcept;

 // 23.806.0.1, modifiers:
 template <class... Args> void emplace_back(Args&&... args);
 void push_back(const T& x);
 void push_back(T&& x);
```

```
    void pop_back();

    template <class... Args> iterator emplace(const_iterator position, Args&&... args);
    iterator insert(const_iterator position, const T& x);
    iterator     insert(const_iterator position, T&& x);
    iterator     insert(const_iterator position, size_type n, const T& x);
    template <class InputIterator>
        iterator insert(const_iterator position,
                        InputIterator first, InputIterator last);
    iterator     insert(const_iterator position, initializer_list<T> il);
    iterator erase(const_iterator position);
    iterator erase(const_iterator first, const_iterator last);
    void     swap(vector&);
    void     clear() noexcept;
  };

  template <class T, class Allocator>
    bool operator==(const vector<T,Allocator>& x, const vector<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator< (const vector<T,Allocator>& x, const vector<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator!=(const vector<T,Allocator>& x, const vector<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator> (const vector<T,Allocator>& x, const vector<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator>=(const vector<T,Allocator>& x, const vector<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator<=(const vector<T,Allocator>& x, const vector<T,Allocator>& y);

  // 23.890.0.1, specialized algorithms:
  template <class T, class Allocator>
    void swap(vector<T,Allocator>& x, vector<T,Allocator>& y);
}
```

23.748.0.1 `vector` constructors, copy, and assignment [vector.cons]

`explicit vector(const Allocator&);`

1 *Effects:* Constructs an empty `vector`, using the specified allocator.

2 *Complexity:* Constant.

`explicit vector(size_type n, const Allocator& = Allocator());`

1 *Effects:* Constructs a `vector` with `n` default-inserted elements using the specified allocator.

1 *Requires:* `T` shall be `DefaultInsertable` into `*this`.

2 *Complexity:* Linear in `n`.

```
vector(size_type n, const T& value,
       const Allocator& = Allocator());
```

3 *Effects:* Constructs a `vector` with `n` copies of `value`, using the specified allocator.

4 *Requires:* `T` shall be `CopyInsertable` into `*this`.

5 *Complexity:* Linear in `n`.

```
template <class InputIterator>
  vector(InputIterator first, InputIterator last,
         const Allocator& = Allocator());
```

6

Effects: Constructs a `vector` equal to the range `[first,last)`, using the specified allocator.

1

Complexity: Makes only N calls to the copy constructor of `T` (where N is the distance between `first` and `last`) and no reallocations if iterators first and last are of forward, bidirectional, or random access categories. It makes order `N` calls to the copy constructor of `T` and order $\log(N)$ reallocations if they are just input iterators.

23.758.0.1 `vector` capacity [vector.capacity]

```
size_type capacity() const noexcept;
```

1

Returns: The total number of elements that the vector can hold without requiring reallocation.

```
void reserve(size_type n);
```

1 *Requires:* `T` shall be `MoveInsertable` into `*this`.

2

Effects: A directive that informs a `vector` of a planned change in size, so that it can manage the storage allocation accordingly. After `reserve()`, `capacity()` is greater or equal to the argument of `reserve` if reallocation happens; and equal to the previous value of `capacity()` otherwise. Reallocation happens at this point if and only if the current capacity is less than the argument of `reserve()`. If an exception is thrown other than by the move constructor of a non-`CopyInsertable` type, there are no effects.

1

Complexity: It does not change the size of the sequence and takes at most linear time in the size of the sequence.

1

Throws: `length_error` if `n > max_size()`.[266]

1

Remarks: Reallocation invalidates all the references, pointers, and iterators referring to the elements in the sequence. No reallocation shall take place during insertions that happen after a call to `reserve()` until the time when an insertion would make the size of the vector greater than the value of `capacity()`.

```
void shrink_to_fit();
```

1 *Requires:* `T` shall be `MoveInsertable` into `*this`.

2 *Complexity:* Linear in the size of the sequence.

3

Remarks: `shrink_to_fit` is a non-binding request to reduce `capacity()` to `size()`. [*Note:* The request is non-binding to allow latitude for implementation-specific optimizations. — *end note*] If an exception is thrown other than by the move constructor of a non-`CopyInsertable` `T` there are no effects.

266) `reserve()` uses `Allocator::allocate()` which may throw an appropriate exception.

```
void swap(vector& x);
```

1 *Effects:* Exchanges the contents and `capacity()` of `*this` with that of `x`.

2 *Complexity:* Constant time.

```
void resize(size_type sz);
```

3 *Effects:* If `sz <= size()`, equivalent to calling `pop_back() size() - sz` times. If `size() < sz`, appends `sz - size()` default-inserted elements to the sequence.

4 *Requires:* `T` shall be `MoveInsertable` and `DefaultInsertable` into `*this`.

5 *Remarks:* If an exception is thrown other than by the move constructor of a non-`CopyInsertable` `T` there are no effects.

```
void resize(size_type sz, const T& c);
```

6 *Effects:* If `sz <= size()`, equivalent to calling `pop_back() size() - sz` times. If `size() < sz`, appends `sz - size()` copies of `c` to the sequence.

7 *Requires:* `T` shall be `CopyInsertable` into `*this`.

8 *Remarks:* If an exception is thrown there are no effects.

23.801.0.1 `vector` data [vector.data]

```
T*         data() noexcept;
const T*   data() const noexcept;
```

1

Returns: A pointer such that `[data(),data() + size())` is a valid range. For a non-empty vector, `data() == &front()`.

1 *Complexity:* Constant time.

23.806.0.1 `vector` modifiers [vector.modifiers]

```
iterator insert(const_iterator position, const T& x);
iterator insert(const_iterator position, T&& x);
iterator insert(const_iterator position, size_type n, const T& x);
template <class InputIterator>
  iterator insert(const_iterator position, InputIterator first, InputIterator last);
iterator insert(const_iterator position, initializer_list<T>);

template <class... Args> void emplace_back(Args&&... args);
template <class... Args> iterator emplace(const_iterator position, Args&&... args);
void push_back(const T& x);
void push_back(T&& x);
```

1

Remarks: Causes reallocation if the new size is greater than the old capacity. If no reallocation happens, all the iterators and references before the insertion point remain valid. If an exception is thrown other than by the copy constructor, move constructor, assignment operator, or move assignment operator of `T` or by any `InputIterator` operation there are no effects. If an exception is thrown while inserting a single element at the end and `T` is `CopyInsertable` or `is_nothrow_move_constructible<T>::value` is `true`, there are no effects. Otherwise, if an exception is thrown by the move constructor of a non-`CopyInsertable` `T`, the effects are unspecified.

1

Complexity: The complexity is linear in the number of elements inserted plus the distance to the end of the vector.

```
iterator erase(const_iterator position);
iterator erase(const_iterator first, const_iterator last);
```

1

Effects: Invalidates iterators and references at or after the point of the erase.

1

Complexity: The destructor of `T` is called the number of times equal to the number of the elements erased, but the move assignment operator of `T` is called the number of times equal to the number of elements in the vector after the erased elements.

1 *Throws:* Nothing unless an exception is thrown by the copy constructor, move constructor, assignment operator, or move assignment operator of `T`.

23.890.0.1 `vector` specialized algorithms [vector.special]

```
template <class T, class Allocator>
  void swap(vector<T,Allocator>& x, vector<T,Allocator>& y);
```

1 *Effects:*

```
x.swap(y);
```

23.890.1 Class `vector<bool>` [vector.bool]

1

To optimize space allocation, a specialization of vector for `bool` elements is provided:

```
namespace std {
  template <class Allocator> class vector<bool, Allocator> {
  public:
    // types:
    typedef bool                                    const_reference;
    typedef implementation-defined                  iterator;       // see 23.2
    typedef implementation-defined                  const_iterator; // see 23.2
    typedef implementation-defined                  size_type;      // see 23.2
    typedef implementation-defined                  difference_type;// see 23.2
    typedef bool                                    value_type;
    typedef Allocator                               allocator_type;
    typedef implementation-defined                  pointer;
    typedef implementation-defined                  const_pointer;
    typedef std::reverse_iterator<iterator>         reverse_iterator;
    typedef std::reverse_iterator<const_iterator>   const_reverse_iterator;

    // bit reference:
    class reference {
      friend class vector;
      reference() noexcept;
    public:
      ~reference();
      operator bool() const noexcept;
      reference& operator=(const bool x) noexcept;
      reference& operator=(const reference& x) noexcept;
      void flip() noexcept;     // flips the bit
```

```
};

// construct/copy/destroy:
vector() : vector(Allocator()) { }
explicit vector(const Allocator&);
explicit vector(size_type n, const Allocator& = Allocator());
vector(size_type n, const bool& value,
       const Allocator& = Allocator());
template <class InputIterator>
  vector(InputIterator first, InputIterator last,
         const Allocator& = Allocator());
vector(const vector<bool,Allocator>& x);
vector(vector<bool,Allocator>&& x);
vector(const vector&, const Allocator&);
vector(vector&&, const Allocator&);
vector(initializer_list<bool>, const Allocator& = Allocator()));
~vector();
vector<bool,Allocator>& operator=(const vector<bool,Allocator>& x);
vector<bool,Allocator>& operator=(vector<bool,Allocator>&& x);
vector& operator=(initializer_list<bool>);
template <class InputIterator>
  void assign(InputIterator first, InputIterator last);
void assign(size_type n, const bool& t);
void assign(initializer_list<bool>);
allocator_type get_allocator() const noexcept;

// iterators:
iterator               begin() noexcept;
const_iterator         begin() const noexcept;
iterator               end() noexcept;
const_iterator         end() const noexcept;
reverse_iterator       rbegin() noexcept;
const_reverse_iterator rbegin() const noexcept;
reverse_iterator       rend() noexcept;
const_reverse_iterator rend() const noexcept;

const_iterator         cbegin() const noexcept;
const_iterator         cend() const noexcept;
const_reverse_iterator crbegin() const noexcept;
const_reverse_iterator crend() const noexcept;

// capacity:
size_type size() const noexcept;
size_type max_size() const noexcept;
void      resize(size_type sz, bool c = false);
size_type capacity() const noexcept;
bool      empty() const noexcept;
void      reserve(size_type n);
void      shrink_to_fit();

// element access:
reference       operator[](size_type n);
const_reference operator[](size_type n) const;
const_reference at(size_type n) const;
reference       at(size_type n);
```

```
    reference       front();
    const_reference front() const;
    reference       back();
    const_reference back() const;

    // modifiers:
    template <class... Args> void emplace_back(Args&&... args);
    void push_back(const bool& x);
    void pop_back();
    template <class... Args> iterator emplace(const_iterator position, Args&&... args);
    iterator insert(const_iterator position, const bool& x);
    iterator insert (const_iterator position, size_type n, const bool& x);
    template <class InputIterator>
        iterator insert(const_iterator position,
                        InputIterator first, InputIterator last);
    iterator insert(const_iterator position, initializer_list<bool> il);

    iterator erase(const_iterator position);
    iterator erase(const_iterator first, const_iterator last);
    void swap(vector<bool,Allocator>&);
    static void swap(reference x, reference y) noexcept;
    void flip() noexcept;       // flips all bits
    void clear() noexcept;
  };
}
```

1

Unless described below, all operations have the same requirements and semantics as the primary `vector` template, except that operations dealing with the `bool` value type map to bit values in the container storage and allocator_traits::construct (20.89.0.1) is not used to construct these values.

1

There is no requirement that the data be stored as a contiguous allocation of `bool` values. A space-optimized representation of bits is recommended instead.

1 `reference` is a class that simulates the behavior of references of a single bit in `vector<bool>`. The conversion operator returns `true` when the bit is set, and `false` otherwise. The assignment operator sets the bit when the argument is (convertible to) `true` and clears it otherwise. `flip` reverses the state of the bit.

`void flip() noexcept;`

2 *Effects:* Replaces each element in the container with its complement.

`static void swap(reference x, reference y) noexcept;`

3 *Effects:* exchanges the contents of `x` and `y` as if by

```
bool b = x;
x = y;
y = b;
```

`template <class Allocator> struct hash<vector<bool, Allocator> >;`

4 The template specialization shall meet the requirements of class template `hash` (20.443.1).

23.894 Associative containers [associative]

23.894.1 In general [associative.general]

1 The header `<map>` defines the class templates `map` and `multimap`; the header `<set>` defines the class templates `set` and `multiset`.

23.894.2 Header `<map>` synopsis [associative.map.syn]

```
#include <initializer_list>

namespace std {

  template <class Key, class T, class Compare = less<Key>,
            class Allocator = allocator<pair<const Key, T> > >
    class map;
  template <class Key, class T, class Compare, class Allocator>
    bool operator==(const map<Key,T,Compare,Allocator>& x,
                    const map<Key,T,Compare,Allocator>& y);
  template <class Key, class T, class Compare, class Allocator>
    bool operator< (const map<Key,T,Compare,Allocator>& x,
                    const map<Key,T,Compare,Allocator>& y);
  template <class Key, class T, class Compare, class Allocator>
    bool operator!=(const map<Key,T,Compare,Allocator>& x,
                    const map<Key,T,Compare,Allocator>& y);
  template <class Key, class T, class Compare, class Allocator>
    bool operator> (const map<Key,T,Compare,Allocator>& x,
                    const map<Key,T,Compare,Allocator>& y);
  template <class Key, class T, class Compare, class Allocator>
    bool operator>=(const map<Key,T,Compare,Allocator>& x,
                    const map<Key,T,Compare,Allocator>& y);
  template <class Key, class T, class Compare, class Allocator>
    bool operator<=(const map<Key,T,Compare,Allocator>& x,
                    const map<Key,T,Compare,Allocator>& y);
  template <class Key, class T, class Compare, class Allocator>
    void swap(map<Key,T,Compare,Allocator>& x,
              map<Key,T,Compare,Allocator>& y);

  template <class Key, class T, class Compare = less<Key>,
            class Allocator = allocator<pair<const Key, T> > >
    class multimap;
  template <class Key, class T, class Compare, class Allocator>
    bool operator==(const multimap<Key,T,Compare,Allocator>& x,
                    const multimap<Key,T,Compare,Allocator>& y);
  template <class Key, class T, class Compare, class Allocator>
    bool operator< (const multimap<Key,T,Compare,Allocator>& x,
                    const multimap<Key,T,Compare,Allocator>& y);
  template <class Key, class T, class Compare, class Allocator>
    bool operator!=(const multimap<Key,T,Compare,Allocator>& x,
                    const multimap<Key,T,Compare,Allocator>& y);
  template <class Key, class T, class Compare, class Allocator>
    bool operator> (const multimap<Key,T,Compare,Allocator>& x,
                    const multimap<Key,T,Compare,Allocator>& y);
  template <class Key, class T, class Compare, class Allocator>
    bool operator>=(const multimap<Key,T,Compare,Allocator>& x,
                    const multimap<Key,T,Compare,Allocator>& y);
  template <class Key, class T, class Compare, class Allocator>
```

```
    bool operator<=(const multimap<Key,T,Compare,Allocator>& x,
                    const multimap<Key,T,Compare,Allocator>& y);
  template <class Key, class T, class Compare, class Allocator>
    void swap(multimap<Key,T,Compare,Allocator>& x,
              multimap<Key,T,Compare,Allocator>& y);
}
```

23.894.3 Header <set> synopsis [associative.set.syn]

```
#include <initializer_list>

namespace std {

  template <class Key, class Compare = less<Key>,
            class Allocator = allocator<Key> >
    class set;
  template <class Key, class Compare, class Allocator>
    bool operator==(const set<Key,Compare,Allocator>& x,
                    const set<Key,Compare,Allocator>& y);
  template <class Key, class Compare, class Allocator>
    bool operator< (const set<Key,Compare,Allocator>& x,
                    const set<Key,Compare,Allocator>& y);
  template <class Key, class Compare, class Allocator>
    bool operator!=(const set<Key,Compare,Allocator>& x,
                    const set<Key,Compare,Allocator>& y);
  template <class Key, class Compare, class Allocator>
    bool operator> (const set<Key,Compare,Allocator>& x,
                    const set<Key,Compare,Allocator>& y);
  template <class Key, class Compare, class Allocator>
    bool operator>=(const set<Key,Compare,Allocator>& x,
                    const set<Key,Compare,Allocator>& y);
  template <class Key, class Compare, class Allocator>
    bool operator<=(const set<Key,Compare,Allocator>& x,
                    const set<Key,Compare,Allocator>& y);
  template <class Key, class Compare, class Allocator>
    void swap(set<Key,Compare,Allocator>& x,
              set<Key,Compare,Allocator>& y);

  template <class Key, class Compare = less<Key>,
            class Allocator = allocator<Key> >
    class multiset;
  template <class Key, class Compare, class Allocator>
    bool operator==(const multiset<Key,Compare,Allocator>& x,
                    const multiset<Key,Compare,Allocator>& y);
  template <class Key, class Compare, class Allocator>
    bool operator< (const multiset<Key,Compare,Allocator>& x,
                    const multiset<Key,Compare,Allocator>& y);
  template <class Key, class Compare, class Allocator>
    bool operator!=(const multiset<Key,Compare,Allocator>& x,
                    const multiset<Key,Compare,Allocator>& y);
  template <class Key, class Compare, class Allocator>
    bool operator> (const multiset<Key,Compare,Allocator>& x,
                    const multiset<Key,Compare,Allocator>& y);
  template <class Key, class Compare, class Allocator>
    bool operator>=(const multiset<Key,Compare,Allocator>& x,
                    const multiset<Key,Compare,Allocator>& y);
```

```
  template <class Key, class Compare, class Allocator>
    bool operator<=(const multiset<Key,Compare,Allocator>& x,
                    const multiset<Key,Compare,Allocator>& y);
  template <class Key, class Compare, class Allocator>
    void swap(multiset<Key,Compare,Allocator>& x,
              multiset<Key,Compare,Allocator>& y);
}
```

23.894.4 Class template map [map]

23.894.4.1 Class template map overview [map.overview]

1 A map is an associative container that supports unique keys (contains at most one of each key value) and provides for fast retrieval of values of another type T based on the keys. The map class supports bidirectional iterators.

2

A map satisfies all of the requirements of a container, of a reversible container (23.2), of an associative container (23.294.1), and of an allocator-aware container (Table 99). A map also provides most operations described in (23.294.1) for unique keys. This means that a map supports the a_uniq operations in (23.294.1) but not the a_eq operations. For a map<Key,T> the key_type is Key and the value_type is pair<const Key,T>. Descriptions are provided here only for operations on map that are not described in one of those tables or for operations where there is additional semantic information.

```
namespace std {
  template <class Key, class T, class Compare = less<Key>,
            class Allocator = allocator<pair<const Key, T> > >
  class map {
  public:
    // types:
    typedef Key                                    key_type;
    typedef T                                      mapped_type;
    typedef pair<const Key, T>                     value_type;
    typedef Compare                                key_compare;
    typedef Allocator                              allocator_type;
    typedef value_type&                            reference;
    typedef const value_type&                      const_reference;
    typedef implementation-defined                 iterator;       // see 23.2
    typedef implementation-defined                 const_iterator; // see 23.2
    typedef implementation-defined                 size_type;      // see 23.2
    typedef implementation-defined                 difference_type;// see 23.2
    typedef typename allocator_traits<Allocator>::pointer           pointer;
    typedef typename allocator_traits<Allocator>::const_pointer     const_pointer;
    typedef std::reverse_iterator<iterator>        reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

    class value_compare {
    friend class map;
    protected:
      Compare comp;
      value_compare(Compare c) : comp(c) {}
    public:
      typedef bool result_type;
      typedef value_type first_argument_type;
      typedef value_type second_argument_type;
      bool operator()(const value_type& x, const value_type& y) const {
```

```
    return comp(x.first, y.first);
  }
};

// 23.896.0.1, construct/copy/destroy:
map() : map(Compare()) { }
explicit map(const Compare& comp,
             const Allocator& = Allocator());
template <class InputIterator>
  map(InputIterator first, InputIterator last,
      const Compare& comp = Compare(), const Allocator& = Allocator());
map(const map& x);
map(map&& x);
explicit map(const Allocator&);
map(const map&, const Allocator&);
map(map&&, const Allocator&);
map(initializer_list<value_type>,
  const Compare& = Compare(),
  const Allocator& = Allocator());
template <class InputIterator>
map(InputIterator first, InputIterator last, const Allocator& a)
  : map(first, last, Compare(), a) { }
map(initializer_list<value_type> il, const Allocator& a)
  : map(il, Compare(), a) { }
~map();
map& operator=(const map& x);
map& operator=(map&& x);
map& operator=(initializer_list<value_type>);
allocator_type get_allocator() const noexcept;

// iterators:
iterator               begin() noexcept;
const_iterator         begin() const noexcept;
iterator               end() noexcept;
const_iterator         end() const noexcept;

reverse_iterator       rbegin() noexcept;
const_reverse_iterator rbegin() const noexcept;
reverse_iterator       rend() noexcept;
const_reverse_iterator rend() const noexcept;

const_iterator         cbegin() const noexcept;
const_iterator         cend() const noexcept;
const_reverse_iterator crbegin() const noexcept;
const_reverse_iterator crend() const noexcept;

// capacity:
bool      empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;

// 23.896.0.2, element access:
T& operator[](const key_type& x);
T& operator[](key_type&& x);
T&       at(const key_type& x);
```

```
  const T& at(const key_type& x) const;

  // 23.924.0.1, modifiers:
  template <class... Args> pair<iterator, bool> emplace(Args&&... args);
  template <class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
  pair<iterator, bool> insert(const value_type& x);
  template <class P> pair<iterator, bool> insert(P&& x);
  iterator insert(const_iterator position, const value_type& x);
  template <class P>
    iterator insert(const_iterator position, P&&);
  template <class InputIterator>
    void insert(InputIterator first, InputIterator last);
  void insert(initializer_list<value_type>);

  iterator  erase(const_iterator position);
  size_type erase(const key_type& x);
  iterator  erase(const_iterator first, const_iterator last);
  void swap(map&);
  void clear() noexcept;

  // observers:
  key_compare   key_comp() const;
  value_compare value_comp() const;

  // map operations:
  iterator       find(const key_type& x);
  const_iterator find(const key_type& x) const;
  template <class K> iterator       find(const K& x);
  template <class K> const_iterator find(const K& x) const;

  size_type      count(const key_type& x) const;
  template <class K> size_type count(const K& x) const;

  iterator       lower_bound(const key_type& x);
  const_iterator lower_bound(const key_type& x) const;
  template <class K> iterator       lower_bound(const K& x);
  template <class K> const_iterator lower_bound(const K& x) const;

  iterator       upper_bound(const key_type& x);
  const_iterator upper_bound(const key_type& x) const;
  template <class K> iterator       upper_bound(const K& x);
  template <class K> const_iterator upper_bound(const K& x) const;

  pair<iterator,iterator>
    equal_range(const key_type& x);
  pair<const_iterator,const_iterator>
    equal_range(const key_type& x) const;
  template <class K>
    pair<iterator, iterator>             equal_range(const K& x);
  template <class K>
    pair<const_iterator, const_iterator> equal_range(const K& x) const;
};

template <class Key, class T, class Compare, class Allocator>
  bool operator==(const map<Key,T,Compare,Allocator>& x,
```

```
                    const map<Key,T,Compare,Allocator>& y);
  template <class Key, class T, class Compare, class Allocator>
    bool operator< (const map<Key,T,Compare,Allocator>& x,
                    const map<Key,T,Compare,Allocator>& y);
  template <class Key, class T, class Compare, class Allocator>
    bool operator!=(const map<Key,T,Compare,Allocator>& x,
                    const map<Key,T,Compare,Allocator>& y);
  template <class Key, class T, class Compare, class Allocator>
    bool operator> (const map<Key,T,Compare,Allocator>& x,
                    const map<Key,T,Compare,Allocator>& y);
  template <class Key, class T, class Compare, class Allocator>
    bool operator>=(const map<Key,T,Compare,Allocator>& x,
                    const map<Key,T,Compare,Allocator>& y);
  template <class Key, class T, class Compare, class Allocator>
    bool operator<=(const map<Key,T,Compare,Allocator>& x,
                    const map<Key,T,Compare,Allocator>& y);

  // specialized algorithms:
  template <class Key, class T, class Compare, class Allocator>
    void swap(map<Key,T,Compare,Allocator>& x,
              map<Key,T,Compare,Allocator>& y);
}
```

23.896.0.1 `map` constructors, copy, and assignment [map.cons]

```
explicit map(const Compare& comp,
             const Allocator& = Allocator());
```

1 *Effects:* Constructs an empty `map` using the specified comparison object and allocator.

2 *Complexity:* Constant.

```
template <class InputIterator>
  map(InputIterator first, InputIterator last,
      const Compare& comp = Compare(), const Allocator& = Allocator());
```

3 *Requires:* If the iterator's indirection operator returns an lvalue or a const rvalue `pair<key_type, mapped_type>`, then both `key_type` and `mapped_type` shall be `CopyInsertable` into `*this`.

4 *Effects:* Constructs an empty `map` using the specified comparison object and allocator, and inserts elements from the range `[first,last)`.

5 *Complexity:* Linear in N if the range `[first,last)` is already sorted using `comp` and otherwise $N \log N$, where N is `last - first`.

23.896.0.2 `map` element access [map.access]

```
T& operator[](const key_type& x);
```

1

Effects: If there is no key equivalent to `x` in the map, inserts `value_type(x, T())` into the map.

1

Requires: `key_type` shall be `CopyInsertable` and `mapped_type` shall be `DefaultInsertable` into `*this`.

1 *Returns:* A reference to the `mapped_type` corresponding to `x` in `*this`.

2 *Complexity:* Logarithmic.

```
T& operator[](key_type&& x);
```

3

Effects: If there is no key equivalent to `x` in the map, inserts `value_type(std::move(x), T())` into the map.

1 *Requires:* `mapped_type` shall be `DefaultInsertable` into `*this`.

2 *Returns:* A reference to the `mapped_type` corresponding to `x` in `*this`.

3

Complexity: Logarithmic.

```
T&       at(const key_type& x);
const T& at(const key_type& x) const;
```

1 *Returns:* A reference to the `mapped_type` corresponding to `x` in `*this`.

2

Throws: An exception object of type `out_of_range` if no such element is present.

1 *Complexity:* Logarithmic.

23.924.0.1 `map` modifiers [map.modifiers]

```
template <class P> pair<iterator, bool> insert(P&& x);
template <class P> iterator insert(const_iterator position, P&& x);
template <class InputIterator>
  void insert(InputIterator first, InputIterator last);
```

1

Effects: The first form is equivalent to `return emplace(std::forward<P>(x))`. The second form is equivalent to `return emplace_hint(position, std::forward<P>(x))`.

1 *Remarks:* These signatures shall not participate in overload resolution unless `std::is_constructible<value_type, P&&>::value` is `true`.

23.925.0.1 `map` specialized algorithms [map.special]

```
template <class Key, class T, class Compare, class Allocator>
  void swap(map<Key,T,Compare,Allocator>& x,
            map<Key,T,Compare,Allocator>& y);
```

1 *Effects:*

```
x.swap(y);
```

23.925.1 Class template `multimap` [multimap]

23.925.1.1 Class template `multimap` overview [multimap.overview]

1

A `multimap` is an associative container that supports equivalent keys (possibly containing multiple copies of the same key value) and provides for fast retrieval of values of another type `T` based on the keys. The `multimap` class supports bidirectional iterators.

1 A `multimap` satisfies all of the requirements of a container and of a reversible container (23.2), of an associative container (23.294.1), and of an allocator-aware container (Table 99). A `multimap` also provides most operations described in (23.294.1) for equal keys. This means that a `multimap` supports the `a_eq` operations in (23.294.1) but not the `a_uniq` operations. For a `multimap<Key,T>` the `key_type` is `Key` and

the `value_type` is `pair<const Key,T>`. Descriptions are provided here only for operations on `multimap` that are not described in one of those tables or for operations where there is additional semantic information.

```
namespace std {
  template <class Key, class T, class Compare = less<Key>,
            class Allocator = allocator<pair<const Key, T> > >
  class multimap {
  public:
    // types:
    typedef Key                                  key_type;
    typedef T                                    mapped_type;
    typedef pair<const Key,T>                    value_type;
    typedef Compare                              key_compare;
    typedef Allocator                            allocator_type;
    typedef value_type&                          reference;
    typedef const value_type&                    const_reference;
    typedef implementation-defined               iterator;       // see 23.2
    typedef implementation-defined               const_iterator; // see 23.2
    typedef implementation-defined               size_type;      // see 23.2
    typedef implementation-defined               difference_type;// see 23.2
    typedef typename allocator_traits<Allocator>::pointer           pointer;
    typedef typename allocator_traits<Allocator>::const_pointer     const_pointer;
    typedef std::reverse_iterator<iterator>      reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

    class value_compare {
    friend class multimap;
    protected:
      Compare comp;
      value_compare(Compare c) : comp(c) { }
    public:
      typedef bool result_type;
      typedef value_type first_argument_type;
      typedef value_type second_argument_type;
      bool operator()(const value_type& x, const value_type& y) const {
        return comp(x.first, y.first);
      }
    };

    // construct/copy/destroy:
    multimap() : multimap(Compare()) { }
    explicit multimap(const Compare& comp,
                      const Allocator& = Allocator());
    template <class InputIterator>
      multimap(InputIterator first, InputIterator last,
               const Compare& comp = Compare(),
               const Allocator& = Allocator());
    multimap(const multimap& x);
    multimap(multimap&& x);
    explicit multimap(const Allocator&);
    multimap(const multimap&, const Allocator&);
    multimap(multimap&&, const Allocator&);
    multimap(initializer_list<value_type>,
      const Compare& = Compare(),
      const Allocator& = Allocator());
    template <class InputIterator>
```

```
  multimap(InputIterator first, InputIterator last, const Allocator& a)
    : multimap(first, last, Compare(), a) { }
  multimap(initializer_list<value_type> il, const Allocator& a)
    : multimap(il, Compare(), a) { }
 ~multimap();
  multimap& operator=(const multimap& x);
  multimap& operator=(multimap&& x);
  multimap& operator=(initializer_list<value_type>);
  allocator_type get_allocator() const noexcept;

  // iterators:
  iterator               begin() noexcept;
  const_iterator         begin() const noexcept;
  iterator               end() noexcept;
  const_iterator         end() const noexcept;

  reverse_iterator       rbegin() noexcept;
  const_reverse_iterator rbegin() const noexcept;
  reverse_iterator       rend() noexcept;
  const_reverse_iterator rend() const noexcept;

  const_iterator         cbegin() const noexcept;
  const_iterator         cend() const noexcept;
  const_reverse_iterator crbegin() const noexcept;
  const_reverse_iterator crend() const noexcept;

  // capacity:
  bool           empty() const noexcept;
  size_type      size() const noexcept;
  size_type      max_size() const noexcept;

  // modifiers:
  template <class... Args> iterator emplace(Args&&... args);
  template <class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
  iterator insert(const value_type& x);
  template <class P> iterator insert(P&& x);
  iterator insert(const_iterator position, const value_type& x);
  template <class P> iterator insert(const_iterator position, P&& x);
  template <class InputIterator>
    void insert(InputIterator first, InputIterator last);
  void insert(initializer_list<value_type>);

  iterator  erase(const_iterator position);
  size_type erase(const key_type& x);
  iterator  erase(const_iterator first, const_iterator last);
  void swap(multimap&);
  void clear() noexcept;

  // observers:
  key_compare    key_comp() const;
  value_compare  value_comp() const;

  // map operations:
  iterator       find(const key_type& x);
  const_iterator find(const key_type& x) const;
```

```
    template <class K> iterator       find(const K& x);
    template <class K> const_iterator find(const K& x) const;

    size_type      count(const key_type& x) const;
    template <class K> size_type count(const K& x) const;

    iterator       lower_bound(const key_type& x);
    const_iterator lower_bound(const key_type& x) const;
    template <class K> iterator       lower_bound(const K& x);
    template <class K> const_iterator lower_bound(const K& x) const;

    iterator       upper_bound(const key_type& x);
    const_iterator upper_bound(const key_type& x) const;
    template <class K> iterator       upper_bound(const K& x);
    template <class K> const_iterator upper_bound(const K& x) const;

    pair<iterator,iterator>
      equal_range(const key_type& x);
    pair<const_iterator,const_iterator>
      equal_range(const key_type& x) const;
    template <class K>
      pair<iterator, iterator>             equal_range(const K& x);
    template <class K>
      pair<const_iterator, const_iterator> equal_range(const K& x) const;
  };

  template <class Key, class T, class Compare, class Allocator>
    bool operator==(const multimap<Key,T,Compare,Allocator>& x,
                    const multimap<Key,T,Compare,Allocator>& y);
  template <class Key, class T, class Compare, class Allocator>
    bool operator< (const multimap<Key,T,Compare,Allocator>& x,
                    const multimap<Key,T,Compare,Allocator>& y);
  template <class Key, class T, class Compare, class Allocator>
    bool operator!=(const multimap<Key,T,Compare,Allocator>& x,
                    const multimap<Key,T,Compare,Allocator>& y);
  template <class Key, class T, class Compare, class Allocator>
    bool operator> (const multimap<Key,T,Compare,Allocator>& x,
                    const multimap<Key,T,Compare,Allocator>& y);
  template <class Key, class T, class Compare, class Allocator>
    bool operator>=(const multimap<Key,T,Compare,Allocator>& x,
                    const multimap<Key,T,Compare,Allocator>& y);
  template <class Key, class T, class Compare, class Allocator>
    bool operator<=(const multimap<Key,T,Compare,Allocator>& x,
                    const multimap<Key,T,Compare,Allocator>& y);

  // specialized algorithms:
  template <class Key, class T, class Compare, class Allocator>
    void swap(multimap<Key,T,Compare,Allocator>& x,
              multimap<Key,T,Compare,Allocator>& y);
}
```

23.928.0.1 multimap constructors [multimap.cons]

```
explicit multimap(const Compare& comp,
                  const Allocator& = Allocator());
```

1 *Effects:* Constructs an empty `multimap` using the specified comparison object and allocator.

2 *Complexity:* Constant.

```
template <class InputIterator>
  multimap(InputIterator first, InputIterator last,
           const Compare& comp = Compare(),
           const Allocator& = Allocator());
```

3 *Requires:* If the iterator's indirection operator returns an lvalue or a const rvalue `pair<key_type, mapped_type>`, then both `key_type` and `mapped_type` shall be `CopyInsertable` into `*this`.

4 *Effects:* Constructs an empty `multimap` using the specified comparison object and allocator, and inserts elements from the range `[first,last)`.

5 *Complexity:* Linear in N if the range `[first,last)` is already sorted using `comp` and otherwise $N \log N$, where N is `last - first`.

23.928.0.2 `multimap` modifiers [multimap.modifiers]

```
template <class P> iterator insert(P&& x);
template <class P> iterator insert(const_iterator position, P&& x);
```

1 *Effects:* The first form is equivalent to `return emplace(std::forward<P>(x))`. The second form is equivalent to `return emplace_hint(position, std::forward<P>(x))`.

2 *Remarks:* These signatures shall not participate in overload resolution unless `std::is_constructible<value_type, P&&>::value` is `true`.

23.928.0.3 `multimap` specialized algorithms [multimap.special]

```
template <class Key, class T, class Compare, class Allocator>
  void swap(multimap<Key,T,Compare,Allocator>& x,
            multimap<Key,T,Compare,Allocator>& y);
```

1 *Effects:*

```
x.swap(y);
```

23.928.1 Class template `set` [set]

23.928.1.1 Class template `set` overview [set.overview]

1

A `set` is an associative container that supports unique keys (contains at most one of each key value) and provides for fast retrieval of the keys themselves. The `set` class supports bidirectional iterators.

1

A `set` satisfies all of the requirements of a container, of a reversible container (23.2), of an associative container (23.294.1), and of an allocator-aware container (Table 99). A `set` also provides most operations described in (23.294.1) for unique keys. This means that a `set` supports the `a_uniq` operations in (23.294.1) but not the `a_eq` operations. For a `set<Key>` both the `key_type` and `value_type` are `Key`. Descriptions are provided here only for operations on `set` that are not described in one of these tables and for operations where there is additional semantic information.

```
namespace std {
  template <class Key, class Compare = less<Key>,
            class Allocator = allocator<Key> >
  class set {
  public:
```

```
// types:
typedef Key                                    key_type;
typedef Key                                    value_type;
typedef Compare                                key_compare;
typedef Compare                                value_compare;
typedef Allocator                              allocator_type;
typedef value_type&                            reference;
typedef const value_type&                      const_reference;
typedef implementation-defined                 iterator;       // See 23.2
typedef implementation-defined                 const_iterator; // See 23.2
typedef implementation-defined                 size_type;      // See 23.2
typedef implementation-defined                 difference_type;// See 23.2
typedef typename allocator_traits<Allocator>::pointer           pointer;
typedef typename allocator_traits<Allocator>::const_pointer     const_pointer;
typedef std::reverse_iterator<iterator>        reverse_iterator;
typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

// 23.931.0.1, construct/copy/destroy:
set() : set(Compare()) { }
explicit set(const Compare& comp,
             const Allocator& = Allocator());
template <class InputIterator>
  set(InputIterator first, InputIterator last,
      const Compare& comp = Compare(), const Allocator& = Allocator());
set(const set& x);
set(set&& x);
explicit set(const Allocator&);
set(const set&, const Allocator&);
set(set&&, const Allocator&);
set(initializer_list<value_type>,
  const Compare& = Compare(),
  const Allocator& = Allocator());
template <class InputIterator>
set(InputIterator first, InputIterator last, const Allocator& a)
  : set(first, last, Compare(), a) { }
set(initializer_list<value_type> il, const Allocator& a)
  : set(il, Compare(), a) { }
~set();
set& operator=(const set& x);
set& operator=(set&& x);
set& operator=(initializer_list<value_type>);
allocator_type get_allocator() const noexcept;

// iterators:
iterator               begin() noexcept;
const_iterator         begin() const noexcept;
iterator               end() noexcept;
const_iterator         end() const noexcept;

reverse_iterator       rbegin() noexcept;
const_reverse_iterator rbegin() const noexcept;
reverse_iterator       rend() noexcept;
const_reverse_iterator rend() const noexcept;

const_iterator         cbegin() const noexcept;
```

```
const_iterator         cend() const noexcept;
const_reverse_iterator crbegin() const noexcept;
const_reverse_iterator crend() const noexcept;

// capacity:
bool          empty() const noexcept;
size_type     size() const noexcept;
size_type     max_size() const noexcept;

// modifiers:
template <class... Args> pair<iterator, bool> emplace(Args&&... args);
template <class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
pair<iterator,bool> insert(const value_type& x);
pair<iterator,bool> insert(value_type&& x);
iterator insert(const_iterator position, const value_type& x);
iterator insert(const_iterator position, value_type&& x);
template <class InputIterator>
  void insert(InputIterator first, InputIterator last);
void insert(initializer_list<value_type>);

iterator  erase(const_iterator position);
size_type erase(const key_type& x);
iterator  erase(const_iterator first, const_iterator last);
void swap(set&);
void clear() noexcept;

// observers:
key_compare   key_comp() const;
value_compare value_comp() const;

// set operations:
iterator        find(const key_type& x);
const_iterator  find(const key_type& x) const;
template <class K> iterator       find(const K& x);
template <class K> const_iterator find(const K& x) const;

size_type count(const key_type& x) const;
template <class K> size_type count(const K& x) const;

iterator        lower_bound(const key_type& x);
const_iterator  lower_bound(const key_type& x) const;
template <class K> iterator       lower_bound(const K& x);
template <class K> const_iterator lower_bound(const K& x) const;

iterator        upper_bound(const key_type& x);
const_iterator  upper_bound(const key_type& x) const;
template <class K> iterator       upper_bound(const K& x);
template <class K> const_iterator upper_bound(const K& x) const;

pair<iterator,iterator>             equal_range(const key_type& x);
pair<const_iterator,const_iterator> equal_range(const key_type& x) const;
template <class K>
  pair<iterator, iterator>             equal_range(const K& x);
template <class K>
  pair<const_iterator, const_iterator> equal_range(const K& x) const;
```

```
  };

  template <class Key, class Compare, class Allocator>
    bool operator==(const set<Key,Compare,Allocator>& x,
                    const set<Key,Compare,Allocator>& y);
  template <class Key, class Compare, class Allocator>
    bool operator< (const set<Key,Compare,Allocator>& x,
                    const set<Key,Compare,Allocator>& y);
  template <class Key, class Compare, class Allocator>
    bool operator!=(const set<Key,Compare,Allocator>& x,
                    const set<Key,Compare,Allocator>& y);
  template <class Key, class Compare, class Allocator>
    bool operator> (const set<Key,Compare,Allocator>& x,
                    const set<Key,Compare,Allocator>& y);
  template <class Key, class Compare, class Allocator>
    bool operator>=(const set<Key,Compare,Allocator>& x,
                    const set<Key,Compare,Allocator>& y);
  template <class Key, class Compare, class Allocator>
    bool operator<=(const set<Key,Compare,Allocator>& x,
                    const set<Key,Compare,Allocator>& y);

  // specialized algorithms:
  template <class Key, class Compare, class Allocator>
    void swap(set<Key,Compare,Allocator>& x,
              set<Key,Compare,Allocator>& y);
}
```

23.931.0.1 set constructors, copy, and assignment [set.cons]

```
explicit set(const Compare& comp,
             const Allocator& = Allocator());
```

1 *Effects:* Constructs an empty set using the specified comparison objects and allocator.

2 *Complexity:* Constant.

```
template <class InputIterator>
  set(InputIterator first, InputIterator last,
      const Compare& comp = Compare(), const Allocator& = Allocator());
```

3 *Effects:* Constructs an empty `set` using the specified comparison object and allocator, and inserts elements from the range `[first,last)`.

4 *Requires:* If the iterator's indirection operator returns an lvalue or a non-const rvalue, then `Key` shall be `CopyInsertable` into `*this`.

5 *Complexity:* Linear in N if the range `[first,last)` is already sorted using `comp` and otherwise $N \log N$, where N is `last - first`.

23.931.0.2 set specialized algorithms [set.special]

```
template <class Key, class Compare, class Allocator>
  void swap(set<Key,Compare,Allocator>& x,
            set<Key,Compare,Allocator>& y);
```

1 *Effects:*

```
x.swap(y);
```

23.931.1 Class template multiset [multiset]

23.931.1.1 Class template multiset overview [multiset.overview]

1 A multiset is an associative container that supports equivalent keys (possibly contains multiple copies of the same key value) and provides for fast retrieval of the keys themselves. The multiset class supports bidirectional iterators.

2 A multiset satisfies all of the requirements of a container, of a reversible container (23.2), of an associative container (23.294.1), and of an allocator-aware container (Table 99). multiset also provides most operations described in (23.294.1) for duplicate keys. This means that a multiset supports the a_eq operations in (23.294.1) but not the a_uniq operations. For a multiset<Key> both the key_type and value_type are Key. Descriptions are provided here only for operations on multiset that are not described in one of these tables and for operations where there is additional semantic information.

```
namespace std {
  template <class Key, class Compare = less<Key>,
            class Allocator = allocator<Key> >
  class multiset {
  public:
    // types:
    typedef Key                                                    key_type;
    typedef Key                                                    value_type;
    typedef Compare                                                key_compare;
    typedef Compare                                                value_compare;
    typedef Allocator                                              allocator_type;
    typedef value_type&                                            reference;
    typedef const value_type&                                      const_reference;
    typedef implementation-defined                iterator;       // see 23.2
    typedef implementation-defined                const_iterator; // see 23.2
    typedef implementation-defined                size_type;      // see 23.2
    typedef implementation-defined                difference_type;// see 23.2
    typedef typename allocator_traits<Allocator>::pointer          pointer;
    typedef typename allocator_traits<Allocator>::const_pointer    const_pointer;
    typedef std::reverse_iterator<iterator>       reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

    // construct/copy/destroy:
    multiset() : multiset(Compare()) { }
    explicit multiset(const Compare& comp,
                      const Allocator& = Allocator());
    template <class InputIterator>
      multiset(InputIterator first, InputIterator last,
               const Compare& comp = Compare(),
               const Allocator& = Allocator());
    multiset(const multiset& x);
    multiset(multiset&& x);
    explicit multiset(const Allocator&);
    multiset(const multiset&, const Allocator&);
    multiset(multiset&&, const Allocator&);
    multiset(initializer_list<value_type>,
      const Compare& = Compare(),
      const Allocator& = Allocator());
    template <class InputIterator>
    multiset(InputIterator first, InputIterator last, const Allocator& a)
      : multiset(first, last, Compare(), a) { }
    multiset(initializer_list<value_type> il, const Allocator& a)
```

```
  : multiset(il, Compare(), a) { }
~multiset();
multiset& operator=(const multiset& x);
multiset& operator=(multiset&& x);
multiset& operator=(initializer_list<value_type>);
allocator_type get_allocator() const noexcept;

// iterators:
iterator               begin() noexcept;
const_iterator         begin() const noexcept;
iterator               end() noexcept;
const_iterator         end() const noexcept;

reverse_iterator       rbegin() noexcept;
const_reverse_iterator rbegin() const noexcept;
reverse_iterator       rend() noexcept;
const_reverse_iterator rend() const noexcept;

const_iterator         cbegin() const noexcept;
const_iterator         cend() const noexcept;
const_reverse_iterator crbegin() const noexcept;
const_reverse_iterator crend() const noexcept;

// capacity:
bool          empty() const noexcept;
size_type     size() const noexcept;
size_type     max_size() const noexcept;

// modifiers:
template <class... Args> iterator emplace(Args&&... args);
template <class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
iterator insert(const value_type& x);
iterator insert(value_type&& x);
iterator insert(const_iterator position, const value_type& x);
iterator insert(const_iterator position, value_type&& x);
template <class InputIterator>
  void insert(InputIterator first, InputIterator last);
void insert(initializer_list<value_type>);

iterator  erase(const_iterator position);
size_type erase(const key_type& x);
iterator  erase(const_iterator first, const_iterator last);
void swap(multiset&);
void clear() noexcept;

// observers:
key_compare   key_comp() const;
value_compare value_comp() const;

// set operations:
iterator        find(const key_type& x);
const_iterator  find(const key_type& x) const;
template <class K> iterator       find(const K& x);
template <class K> const_iterator find(const K& x) const;
```

```
    size_type count(const key_type& x) const;
    template <class K> size_type count(const K& x) const;

    iterator        lower_bound(const key_type& x);
    const_iterator  lower_bound(const key_type& x) const;
    template <class K> iterator       lower_bound(const K& x);
    template <class K> const_iterator lower_bound(const K& x) const;

    iterator        upper_bound(const key_type& x);
    const_iterator  upper_bound(const key_type& x) const;
    template <class K> iterator       upper_bound(const K& x);
    template <class K> const_iterator upper_bound(const K& x) const;

    pair<iterator,iterator>             equal_range(const key_type& x);
    pair<const_iterator,const_iterator> equal_range(const key_type& x) const;
    template <class K>
      pair<iterator, iterator>             equal_range(const K& x);
    template <class K>
      pair<const_iterator, const_iterator> equal_range(const K& x) const;
  };

  template <class Key, class Compare, class Allocator>
    bool operator==(const multiset<Key,Compare,Allocator>& x,
                    const multiset<Key,Compare,Allocator>& y);
  template <class Key, class Compare, class Allocator>
    bool operator< (const multiset<Key,Compare,Allocator>& x,
                    const multiset<Key,Compare,Allocator>& y);
  template <class Key, class Compare, class Allocator>
    bool operator!=(const multiset<Key,Compare,Allocator>& x,
                    const multiset<Key,Compare,Allocator>& y);
  template <class Key, class Compare, class Allocator>
    bool operator> (const multiset<Key,Compare,Allocator>& x,
                    const multiset<Key,Compare,Allocator>& y);
  template <class Key, class Compare, class Allocator>
    bool operator>=(const multiset<Key,Compare,Allocator>& x,
                    const multiset<Key,Compare,Allocator>& y);
  template <class Key, class Compare, class Allocator>
    bool operator<=(const multiset<Key,Compare,Allocator>& x,
                    const multiset<Key,Compare,Allocator>& y);

  // specialized algorithms:
  template <class Key, class Compare, class Allocator>
    void swap(multiset<Key,Compare,Allocator>& x,
              multiset<Key,Compare,Allocator>& y);
}
```

23.931.1.2 `multiset` constructors [multiset.cons]

```
explicit multiset(const Compare& comp,
                  const Allocator& = Allocator());
```

1 *Effects:* Constructs an empty set using the specified comparison object and allocator.

2 *Complexity:* Constant.

```
template <class InputIterator>
```

```
multiset(InputIterator first, InputIterator last,
         const Compare& comp = Compare(), const Allocator& = Allocator());
```

3 *Requires:* If the iterator's indirection operator returns an lvalue or a const rvalue, then `Key` shall be `CopyInsertable` into `*this`.

4 *Effects:* Constructs an empty `multiset` using the specified comparison object and allocator, and inserts elements from the range `[first,last)`.

5 *Complexity:* Linear in N if the range `[first,last)` is already sorted using `comp` and otherwise $N \log N$, where N is `last - first`.

23.931.1.3 `multiset` specialized algorithms [multiset.special]

```
template <class Key, class Compare, class Allocator>
  void swap(multiset<Key,Compare,Allocator>& x,
            multiset<Key,Compare,Allocator>& y);
```

1 *Effects:*

```
x.swap(y);
```

23.932 Unordered associative containers [unord]

23.932.1 In general [unord.general]

1 The header `<unordered_map>` defines the class templates `unordered_map` and `unordered_multimap`; the header `<unordered_set>` defines the class templates `unordered_set` and `unordered_multiset`.

23.932.2 Header `<unordered_map>` synopsis [unord.map.syn]

```
#include <initializer_list>

namespace std {

  // 23.932.4, class template unordered_map:
  template <class Key,
            class T,
            class Hash = hash<Key>,
            class Pred = std::equal_to<Key>,
            class Alloc = std::allocator<std::pair<const Key, T> > >
    class unordered_map;

  // 23.936.1, class template unordered_multimap:
  template <class Key,
            class T,
            class Hash = hash<Key>,
            class Pred = std::equal_to<Key>,
            class Alloc = std::allocator<std::pair<const Key, T> > >
    class unordered_multimap;

  template <class Key, class T, class Hash, class Pred, class Alloc>
    void swap(unordered_map<Key, T, Hash, Pred, Alloc>& x,
              unordered_map<Key, T, Hash, Pred, Alloc>& y);

  template <class Key, class T, class Hash, class Pred, class Alloc>
    void swap(unordered_multimap<Key, T, Hash, Pred, Alloc>& x,
              unordered_multimap<Key, T, Hash, Pred, Alloc>& y);
```

```
  template <class Key, class T, class Hash, class Pred, class Alloc>
    bool operator==(const unordered_map<Key, T, Hash, Pred, Alloc>& a,
                    const unordered_map<Key, T, Hash, Pred, Alloc>& b);
  template <class Key, class T, class Hash, class Pred, class Alloc>
    bool operator!=(const unordered_map<Key, T, Hash, Pred, Alloc>& a,
                    const unordered_map<Key, T, Hash, Pred, Alloc>& b);
  template <class Key, class T, class Hash, class Pred, class Alloc>
    bool operator==(const unordered_multimap<Key, T, Hash, Pred, Alloc>& a,
                    const unordered_multimap<Key, T, Hash, Pred, Alloc>& b);
  template <class Key, class T, class Hash, class Pred, class Alloc>
    bool operator!=(const unordered_multimap<Key, T, Hash, Pred, Alloc>& a,
                    const unordered_multimap<Key, T, Hash, Pred, Alloc>& b);
} // namespace std
```

23.932.3 Header <unordered_set> synopsis [unord.set.syn]

```
#include <initializer_list>

namespace std {

  // 23.937.1, class template unordered_set:
  template <class Key,
            class Hash = hash<Key>,
            class Pred = std::equal_to<Key>,
            class Alloc = std::allocator<Key> >
    class unordered_set;

  // 23.938.1, class template unordered_multiset:
  template <class Key,
            class Hash = hash<Key>,
            class Pred = std::equal_to<Key>,
            class Alloc = std::allocator<Key> >
    class unordered_multiset;

  template <class Key, class Hash, class Pred, class Alloc>
    void swap(unordered_set<Key, Hash, Pred, Alloc>& x,
              unordered_set<Key, Hash, Pred, Alloc>& y);

  template <class Key, class Hash, class Pred, class Alloc>
    void swap(unordered_multiset<Key, Hash, Pred, Alloc>& x,
              unordered_multiset<Key, Hash, Pred, Alloc>& y);

  template <class Key, class Hash, class Pred, class Alloc>
    bool operator==(const unordered_set<Key, Hash, Pred, Alloc>& a,
                    const unordered_set<Key, Hash, Pred, Alloc>& b);
  template <class Key, class Hash, class Pred, class Alloc>
    bool operator!=(const unordered_set<Key, Hash, Pred, Alloc>& a,
                    const unordered_set<Key, Hash, Pred, Alloc>& b);
  template <class Key, class Hash, class Pred, class Alloc>
    bool operator==(const unordered_multiset<Key, Hash, Pred, Alloc>& a,
                    const unordered_multiset<Key, Hash, Pred, Alloc>& b);
  template <class Key, class Hash, class Pred, class Alloc>
    bool operator!=(const unordered_multiset<Key, Hash, Pred, Alloc>& a,
                    const unordered_multiset<Key, Hash, Pred, Alloc>& b);
} // namespace std
```

23.932.4 Class template unordered_map [unord.map]

23.932.4.1 Class template unordered_map overview [unord.map.overview]

1

An unordered_map is an unordered associative container that supports unique keys (an unordered_map contains at most one of each key value) and that associates values of another type mapped_type with the keys. The unordered_map class supports forward iterators.

1 An unordered_map satisfies all of the requirements of a container, of an unordered associative container, and of an allocator-aware container (Table 99). It provides the operations described in the preceding requirements table for unique keys; that is, an unordered_map supports the a_uniq operations in that table, not the a_eq operations. For an unordered_map<Key, T> the key type is Key, the mapped type is T, and the value type is std::pair<const Key, T>.

2 This section only describes operations on unordered_map that are not described in one of the requirement tables, or for which there is additional semantic information.

```
namespace std {
  template <class Key,
            class T,
            class Hash  = hash<Key>,
            class Pred  = std::equal_to<Key>,
            class Allocator = std::allocator<std::pair<const Key, T> > >
  class unordered_map
  {
  public:
    // types
    typedef Key                                                 key_type;
    typedef std::pair<const Key, T>                             value_type;
    typedef T                                                   mapped_type;
    typedef Hash                                                hasher;
    typedef Pred                                                key_equal;
    typedef Allocator                                           allocator_type;
    typedef typename allocator_traits<Allocator>::pointer       pointer;
    typedef typename allocator_traits<Allocator>::const_pointer const_pointer;
    typedef value_type&                                         reference;
    typedef const value_type&                                   const_reference;
    typedef implementation-defined                              size_type;
    typedef implementation-defined                              difference_type;

    typedef implementation-defined                              iterator;
    typedef implementation-defined                              const_iterator;
    typedef implementation-defined                              local_iterator;
    typedef implementation-defined                              const_local_iterator;

    // construct/destroy/copy
    unordered_map();
    explicit unordered_map(size_type n,
                           const hasher& hf = hasher(),
                           const key_equal& eql = key_equal(),
                           const allocator_type& a = allocator_type());
    template <class InputIterator>
      unordered_map(InputIterator f, InputIterator l,
                    size_type n = see below,
                    const hasher& hf = hasher(),
                    const key_equal& eql = key_equal(),
```

```
                const allocator_type& a = allocator_type());
unordered_map(const unordered_map&);
unordered_map(unordered_map&&);
explicit unordered_map(const Allocator&);
unordered_map(const unordered_map&, const Allocator&);
unordered_map(unordered_map&&, const Allocator&);
unordered_map(initializer_list<value_type>,
  size_type = see below,
  const hasher& hf = hasher(),
  const key_equal& eql = key_equal(),
  const allocator_type& a = allocator_type());
unordered_map(size_type n, const allocator_type& a)
  : unordered_map(n, hasher(), key_equal(), a) { }
unordered_map(size_type n, const hasher& hf, const allocator_type& a)
  : unordered_map(n, hf, key_equal(), a) { }
template <class InputIterator>
  unordered_map(InputIterator f, InputIterator l, size_type n, const allocator_type& a)
  : unordered_map(f, l, n, hasher(), key_equal(), a) { }
template <class InputIterator>
  unordered_map(InputIterator f, InputIterator l, size_type n, const hasher& hf,
  const allocator_type& a)
  : unordered_map(f, l, n, hf, key_equal(), a) { }
unordered_map(initializer_list<value_type> il, size_type n, const allocator_type& a)
  : unordered_map(il, n, hasher(), key_equal(), a) { }
unordered_map(initializer_list<value_type> il, size_type n, const hasher& hf,
const allocator_type& a)
  : unordered_map(il, n, hf, key_equal(), a) { }
~unordered_map();
unordered_map& operator=(const unordered_map&);
unordered_map& operator=(unordered_map&&);
unordered_map& operator=(initializer_list<value_type>);
allocator_type get_allocator() const noexcept;

// size and capacity
bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;

// iterators
iterator       begin() noexcept;
const_iterator begin() const noexcept;
iterator       end() noexcept;
const_iterator end() const noexcept;
const_iterator cbegin() const noexcept;
const_iterator cend() const noexcept;

// modifiers
template <class... Args> pair<iterator, bool> emplace(Args&&... args);
template <class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
pair<iterator, bool> insert(const value_type& obj);
template <class P> pair<iterator, bool> insert(P&& obj);
iterator       insert(const_iterator hint, const value_type& obj);
template <class P> iterator insert(const_iterator hint, P&& obj);
template <class InputIterator> void insert(InputIterator first, InputIterator last);
void insert(initializer_list<value_type>);
```

```
    iterator erase(const_iterator position);
    size_type erase(const key_type& k);
    iterator erase(const_iterator first, const_iterator last);
    void clear() noexcept;

    void swap(unordered_map&);

    // observers
    hasher hash_function() const;
    key_equal key_eq() const;

    // lookup
    iterator       find(const key_type& k);
    const_iterator find(const key_type& k) const;
    size_type count(const key_type& k) const;
    std::pair<iterator, iterator>             equal_range(const key_type& k);
    std::pair<const_iterator, const_iterator> equal_range(const key_type& k) const;

    mapped_type& operator[](const key_type& k);
    mapped_type& operator[](key_type&& k);
    mapped_type& at(const key_type& k);
    const mapped_type& at(const key_type& k) const;

    // bucket interface
    size_type bucket_count() const noexcept;
    size_type max_bucket_count() const noexcept;
    size_type bucket_size(size_type n) const;
    size_type bucket(const key_type& k) const;
    local_iterator begin(size_type n);
    const_local_iterator begin(size_type n) const;
    local_iterator end(size_type n);
    const_local_iterator end(size_type n) const;
    const_local_iterator cbegin(size_type n) const;
    const_local_iterator cend(size_type n) const;

    // hash policy
    float load_factor() const noexcept;
    float max_load_factor() const noexcept;
    void max_load_factor(float z);
    void rehash(size_type n);
    void reserve(size_type n);
  };

  template <class Key, class T, class Hash, class Pred, class Alloc>
    void swap(unordered_map<Key, T, Hash, Pred, Alloc>& x,
              unordered_map<Key, T, Hash, Pred, Alloc>& y);

  template <class Key, class T, class Hash, class Pred, class Alloc>
    bool operator==(const unordered_map<Key, T, Hash, Pred, Alloc>& a,
                    const unordered_map<Key, T, Hash, Pred, Alloc>& b);
  template <class Key, class T, class Hash, class Pred, class Alloc>
    bool operator!=(const unordered_map<Key, T, Hash, Pred, Alloc>& a,
                    const unordered_map<Key, T, Hash, Pred, Alloc>& b);
}
```

23.933.0.1 unordered_map constructors [unord.map.cnstr]

```
unordered_map() : unordered_map(size_type(see below)) { }
explicit unordered_map(size_type n,
                       const hasher& hf = hasher(),
                       const key_equal& eql = key_equal(),
                       const allocator_type& a = allocator_type());
```

1 *Effects:* Constructs an empty `unordered_map` using the specified hash function, key equality function, and allocator, and using at least *n* buckets. For the default constructor, the number of buckets is implementation-defined. `max_load_factor()` returns 1.0.

2 *Complexity:* Constant.

```
template <class InputIterator>
  unordered_map(InputIterator f, InputIterator l,
                size_type n = see below,
                const hasher& hf = hasher(),
                const key_equal& eql = key_equal(),
                const allocator_type& a = allocator_type());
```

3 *Effects:* Constructs an empty `unordered_map` using the specified hash function, key equality function, and allocator, and using at least *n* buckets. If *n* is not provided, the number of buckets is implementation-defined. Then inserts elements from the range `[f, l)`. `max_load_factor()` returns 1.0.

4 *Complexity:* Average case linear, worst case quadratic.

23.933.0.2 unordered_map element access [unord.map.elem]

```
mapped_type& operator[](const key_type& k);
mapped_type& operator[](key_type&& k);
```

1 *Requires:* `mapped_type` shall be `DefaultInsertable` into `*this`. For the first operator, `key_type` shall be `CopyInsertable` into `*this`. For the second operator, `key_type` shall be `MoveConstructible`.

2

Effects: If the `unordered_map` does not already contain an element whose key is equivalent to *k*, the first operator inserts the value `value_type(k, mapped_type())` and the second operator inserts the value `value_type(std::move(k), mapped_type())`.

1

Returns: A reference to `x.second`, where `x` is the (unique) element whose key is equivalent to *k*.

1 *Complexity:* Average case $\mathcal{O}(1)$, worst case $\mathcal{O}$(`size()`).

```
mapped_type& at(const key_type& k);
const mapped_type& at(const key_type& k) const;
```

2 *Returns:* A reference to `x.second`, where `x` is the (unique) element whose key is equivalent to `k`.

3 *Throws:* An exception object of type `out_of_range` if no such element is present.

23.936.0.1 unordered_map modifiers [unord.map.modifiers]

```
template <class P>
  pair<iterator, bool> insert(P&& obj);
```

1 *Effects:* Equivalent to `return emplace(std::forward<P>(obj))`.

2 *Remarks:* This signature shall not participate in overload resolution unless `std::is_constructible<value_type, P&&>::value` is `true`.

```
template <class P>
  iterator insert(const_iterator hint, P&& obj);
```

3 *Effects:* Equivalent to `return emplace_hint(hint, std::forward<P>(obj))`.

4 *Remarks:* This signature shall not participate in overload resolution unless `std::is_constructible<value_type, P&&>::value` is `true`.

23.936.0.2 `unordered_map` swap [unord.map.swap]

```
template <class Key, class T, class Hash, class Pred, class Alloc>
  void swap(unordered_map<Key, T, Hash, Pred, Alloc>& x,
            unordered_map<Key, T, Hash, Pred, Alloc>& y);
```

1 *Effects:* `x.swap(y)`.

23.936.1 Class template `unordered_multimap` [unord.multimap]

23.936.1.1 Class template `unordered_multimap` overview [unord.multimap.overview]

1

An `unordered_multimap` is an unordered associative container that supports equivalent keys (an instance of `unordered_multimap` may contain multiple copies of each key value) and that associates values of another type `mapped_type` with the keys. The `unordered_multimap` class supports forward iterators.

1 An `unordered_multimap` satisfies all of the requirements of a container, of an unordered associative container, and of an allocator-aware container (Table 99). It provides the operations described in the preceding requirements table for equivalent keys; that is, an `unordered_multimap` supports the `a_eq` operations in that table, not the `a_uniq` operations. For an `unordered_multimap<Key, T>` the `key type` is `Key`, the mapped type is `T`, and the value type is `std::pair<const Key, T>`.

2 This section only describes operations on `unordered_multimap` that are not described in one of the requirement tables, or for which there is additional semantic information.

```
namespace std {
  template <class Key,
            class T,
            class Hash  = hash<Key>,
            class Pred  = std::equal_to<Key>,
            class Allocator = std::allocator<std::pair<const Key, T> > >
  class unordered_multimap
  {
  public:
    // types
    typedef Key                                                  key_type;
    typedef std::pair<const Key, T>                              value_type;
    typedef T                                                    mapped_type;
    typedef Hash                                                 hasher;
    typedef Pred                                                 key_equal;
    typedef Allocator                                            allocator_type;
    typedef typename allocator_traits<Allocator>::pointer        pointer;
    typedef typename allocator_traits<Allocator>::const_pointer  const_pointer;
    typedef value_type&                                          reference;
    typedef const value_type&                                    const_reference;
    typedef implementation-defined                               size_type;
    typedef implementation-defined                               difference_type;

    typedef implementation-defined                               iterator;
    typedef implementation-defined                               const_iterator;
```

```
typedef implementation-defined                              local_iterator;
typedef implementation-defined                              const_local_iterator;

// construct/destroy/copy
unordered_multimap();
explicit unordered_multimap(size_type n,
                            const hasher& hf = hasher(),
                            const key_equal& eql = key_equal(),
                            const allocator_type& a = allocator_type());
template <class InputIterator>
  unordered_multimap(InputIterator f, InputIterator l,
                     size_type n = see below,
                     const hasher& hf = hasher(),
                     const key_equal& eql = key_equal(),
                     const allocator_type& a = allocator_type());
unordered_multimap(const unordered_multimap&);
unordered_multimap(unordered_multimap&&);
explicit unordered_multimap(const Allocator&);
unordered_multimap(const unordered_multimap&, const Allocator&);
unordered_multimap(unordered_multimap&&, const Allocator&);
unordered_multimap(initializer_list<value_type>,
  size_type = see below,
  const hasher& hf = hasher(),
  const key_equal& eql = key_equal(),
  const allocator_type& a = allocator_type());
unordered_multimap(size_type n, const allocator_type& a)
  : unordered_multimap(n, hasher(), key_equal(), a) { }
unordered_multimap(size_type n, const hasher& hf, const allocator_type& a)
  : unordered_multimap(n, hf, key_equal(), a) { }
template <class InputIterator>
  unordered_multimap(InputIterator f, InputIterator l, size_type n, const allocator_type& a)
  : unordered_multimap(f, l, n, hasher(), key_equal(), a) { }
template <class InputIterator>
  unordered_multimap(InputIterator f, InputIterator l, size_type n, const hasher& hf,
  const allocator_type& a)
  : unordered_multimap(f, l, n, hf, key_equal(), a) { }
unordered_multimap(initializer_list<value_type> il, size_type n, const allocator_type& a)
  : unordered_multimap(il, n, hasher(), key_equal(), a) { }
unordered_multimap(initializer_list<value_type> il, size_type n, const hasher& hf,
const allocator_type& a)
  : unordered_multimap(il, n, hf, key_equal(), a) { }
~unordered_multimap();
unordered_multimap& operator=(const unordered_multimap&);
unordered_multimap& operator=(unordered_multimap&&);
unordered_multimap& operator=(initializer_list<value_type>);
allocator_type get_allocator() const noexcept;

// size and capacity
bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;

// iterators
iterator       begin() noexcept;
const_iterator begin() const noexcept;
```

```
  iterator       end() noexcept;
  const_iterator end() const noexcept;
  const_iterator cbegin() const noexcept;
  const_iterator cend() const noexcept;

  // modifiers
  template <class... Args> iterator emplace(Args&&... args);
  template <class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
  iterator insert(const value_type& obj);
  template <class P> iterator insert(P&& obj);
  iterator insert(const_iterator hint, const value_type& obj);
  template <class P> iterator insert(const_iterator hint, P&& obj);
  template <class InputIterator> void insert(InputIterator first, InputIterator last);
  void insert(initializer_list<value_type>);

  iterator erase(const_iterator position);
  size_type erase(const key_type& k);
  iterator erase(const_iterator first, const_iterator last);
  void clear() noexcept;

  void swap(unordered_multimap&);

  // observers
  hasher hash_function() const;
  key_equal key_eq() const;

  // lookup
  iterator       find(const key_type& k);
  const_iterator find(const key_type& k) const;
  size_type count(const key_type& k) const;
  std::pair<iterator, iterator>             equal_range(const key_type& k);
  std::pair<const_iterator, const_iterator> equal_range(const key_type& k) const;

  // bucket interface
  size_type bucket_count() const noexcept;
  size_type max_bucket_count() const noexcept;
  size_type bucket_size(size_type n) const;
  size_type bucket(const key_type& k) const;
  local_iterator begin(size_type n);
  const_local_iterator begin(size_type n) const;
  local_iterator end(size_type n);
  const_local_iterator end(size_type n) const;
  const_local_iterator cbegin(size_type n) const;
  const_local_iterator cend(size_type n) const;

  // hash policy
  float load_factor() const noexcept;
  float max_load_factor() const noexcept;
  void max_load_factor(float z);
  void rehash(size_type n);
  void reserve(size_type n);
};

template <class Key, class T, class Hash, class Pred, class Alloc>
  void swap(unordered_multimap<Key, T, Hash, Pred, Alloc>& x,
```

```
                unordered_multimap<Key, T, Hash, Pred, Alloc>& y);

    template <class Key, class T, class Hash, class Pred, class Alloc>
      bool operator==(const unordered_multimap<Key, T, Hash, Pred, Alloc>& a,
                      const unordered_multimap<Key, T, Hash, Pred, Alloc>& b);
    template <class Key, class T, class Hash, class Pred, class Alloc>
      bool operator!=(const unordered_multimap<Key, T, Hash, Pred, Alloc>& a,
                      const unordered_multimap<Key, T, Hash, Pred, Alloc>& b);
  }
```

23.937.0.1 `unordered_multimap` constructors [unord.multimap.cnstr]

```
unordered_multimap() : unordered_multimap(size_type(see below)) { }
explicit unordered_multimap(size_type n,
                            const hasher& hf = hasher(),
                            const key_equal& eql = key_equal(),
                            const allocator_type& a = allocator_type());
```

1 *Effects:* Constructs an empty `unordered_multimap` using the specified hash function, key equality function, and allocator, and using at least *n* buckets. For the default constructor, the number of buckets is implementation-defined. `max_load_factor()` returns 1.0.

2 *Complexity:* Constant.

```
template <class InputIterator>
  unordered_multimap(InputIterator f, InputIterator l,
                     size_type n = see below,
                     const hasher& hf = hasher(),
                     const key_equal& eql = key_equal(),
                     const allocator_type& a = allocator_type());
```

3 *Effects:* Constructs an empty `unordered_multimap` using the specified hash function, key equality function, and allocator, and using at least *n* buckets. If *n* is not provided, the number of buckets is implementation-defined. Then inserts elements from the range `[`*f*`, `*l*`)`. `max_load_factor()` returns 1.0.

4 *Complexity:* Average case linear, worst case quadratic.

23.937.0.2 `unordered_multimap` modifiers [unord.multimap.modifiers]

```
template <class P>
  iterator insert(P&& obj);
```

1 *Effects:* Equivalent to `return emplace(std::forward<P>(obj))`.

2 *Remarks:* This signature shall not participate in overload resolution unless `std::is_constructible<value_type, P&&>::value` is `true`.

```
template <class P>
  iterator insert(const_iterator hint, P&& obj);
```

3 *Effects:* Equivalent to `return emplace_hint(hint, std::forward<P>(obj))`.

4 *Remarks:* This signature shall not participate in overload resolution unless `std::is_constructible<value_type, P&&>::value` is `true`.

23.937.0.3 `unordered_multimap` swap [unord.multimap.swap]

```
template <class Key, class T, class Hash, class Pred, class Alloc>
  void swap(unordered_multimap<Key, T, Hash, Pred, Alloc>& x,
            unordered_multimap<Key, T, Hash, Pred, Alloc>& y);
```

1 *Effects:* `x.swap(y)`.

23.937.1 Class template `unordered_set` [unord.set]

23.937.1.1 Class template `unordered_set` overview [unord.set.overview]

1

An `unordered_set` is an unordered associative container that supports unique keys (an `unordered_set` contains at most one of each key value) and in which the elements' keys are the elements themselves. The `unordered_set` class supports forward iterators.

1 An `unordered_set` satisfies all of the requirements of a container, of an unordered associative container, and of an allocator-aware container (Table 99). It provides the operations described in the preceding requirements table for unique keys; that is, an `unordered_set` supports the `a_uniq` operations in that table, not the `a_eq` operations. For an `unordered_set<Key>` the `key type` and the value type are both `Key`. The `iterator` and `const_iterator` types are both const iterator types. It is unspecified whether they are the same type.

2 This section only describes operations on `unordered_set` that are not described in one of the requirement tables, or for which there is additional semantic information.

```
namespace std {
  template <class Key,
            class Hash  = hash<Key>,
            class Pred  = std::equal_to<Key>,
            class Allocator = std::allocator<Key> >
  class unordered_set
  {
  public:
    // types
    typedef Key                                                  key_type;
    typedef Key                                                  value_type;
    typedef Hash                                                 hasher;
    typedef Pred                                                 key_equal;
    typedef Allocator                                            allocator_type;
    typedef typename allocator_traits<Allocator>::pointer        pointer;
    typedef typename allocator_traits<Allocator>::const_pointer  const_pointer;
    typedef value_type&                                          reference;
    typedef const value_type&                                    const_reference;
    typedef implementation-defined                               size_type;
    typedef implementation-defined                               difference_type;

    typedef implementation-defined                               iterator;
    typedef implementation-defined                               const_iterator;
    typedef implementation-defined                               local_iterator;
    typedef implementation-defined                               const_local_iterator;

    // construct/destroy/copy
    unordered_set();
    explicit unordered_set(size_type n,
                           const hasher& hf = hasher(),
                           const key_equal& eql = key_equal(),
                           const allocator_type& a = allocator_type());
```

```
template <class InputIterator>
  unordered_set(InputIterator f, InputIterator l,
                size_type n = see below,
                const hasher& hf = hasher(),
                const key_equal& eql = key_equal(),
                const allocator_type& a = allocator_type());
unordered_set(const unordered_set&);
unordered_set(unordered_set&&);
explicit unordered_set(const Allocator&);
unordered_set(const unordered_set&, const Allocator&);
unordered_set(unordered_set&&, const Allocator&);
unordered_set(initializer_list<value_type>,
  size_type = see below,
  const hasher& hf = hasher(),
  const key_equal& eql = key_equal(),
  const allocator_type& a = allocator_type());
unordered_set(size_type n, const allocator_type& a)
  : unordered_set(n, hasher(), key_equal(), a) { }
unordered_set(size_type n, const hasher& hf, const allocator_type& a)
  : unordered_set(n, hf, key_equal(), a) { }
template <class InputIterator>
  unordered_set(InputIterator f, InputIterator l, size_type n, const allocator_type& a)
  : unordered_set(f, l, n, hasher(), key_equal(), a) { }
template <class InputIterator>
  unordered_set(InputIterator f, InputIterator l, size_type n, const hasher& hf,
  const allocator_type& a)
  : unordered_set(f, l, n, hf, key_equal(), a) { }
unordered_set(initializer_list<value_type> il, size_type n, const allocator_type& a)
  : unordered_set(il, n, hasher(), key_equal(), a) { }
unordered_set(initializer_list<value_type> il, size_type n, const hasher& hf,
const allocator_type& a)
  : unordered_set(il, n, hf, key_equal(), a) { }
~unordered_set();
unordered_set& operator=(const unordered_set&);
unordered_set& operator=(unordered_set&&);
unordered_set& operator=(initializer_list<value_type>);
allocator_type get_allocator() const noexcept;

// size and capacity
bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;

// iterators
iterator       begin() noexcept;
const_iterator begin() const noexcept;
iterator       end() noexcept;
const_iterator end() const noexcept;
const_iterator cbegin() const noexcept;
const_iterator cend() const noexcept;

// modifiers
template <class... Args> pair<iterator, bool> emplace(Args&&... args);
template <class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
pair<iterator, bool> insert(const value_type& obj);
```

```
    pair<iterator, bool> insert(value_type&& obj);
    iterator insert(const_iterator hint, const value_type& obj);
    iterator insert(const_iterator hint, value_type&& obj);
    template <class InputIterator> void insert(InputIterator first, InputIterator last);
    void insert(initializer_list<value_type>);

    iterator erase(const_iterator position);
    size_type erase(const key_type& k);
    iterator erase(const_iterator first, const_iterator last);
    void clear() noexcept;

    void swap(unordered_set&);

    // observers
    hasher hash_function() const;
    key_equal key_eq() const;

    // lookup
    iterator       find(const key_type& k);
    const_iterator find(const key_type& k) const;
    size_type count(const key_type& k) const;
    std::pair<iterator, iterator>             equal_range(const key_type& k);
    std::pair<const_iterator, const_iterator> equal_range(const key_type& k) const;

    // bucket interface
    size_type bucket_count() const noexcept;
    size_type max_bucket_count() const noexcept;
    size_type bucket_size(size_type n) const;
    size_type bucket(const key_type& k) const;
    local_iterator begin(size_type n);
    const_local_iterator begin(size_type n) const;
    local_iterator end(size_type n);
    const_local_iterator end(size_type n) const;
    const_local_iterator cbegin(size_type n) const;
    const_local_iterator cend(size_type n) const;

    // hash policy
    float load_factor() const noexcept;
    float max_load_factor() const noexcept;
    void max_load_factor(float z);
    void rehash(size_type n);
    void reserve(size_type n);
  };

  template <class Key, class Hash, class Pred, class Alloc>
    void swap(unordered_set<Key, Hash, Pred, Alloc>& x,
              unordered_set<Key, Hash, Pred, Alloc>& y);

  template <class Key, class Hash, class Pred, class Alloc>
    bool operator==(const unordered_set<Key, Hash, Pred, Alloc>& a,
                    const unordered_set<Key, Hash, Pred, Alloc>& b);
  template <class Key, class Hash, class Pred, class Alloc>
    bool operator!=(const unordered_set<Key, Hash, Pred, Alloc>& a,
                    const unordered_set<Key, Hash, Pred, Alloc>& b);
}
```

23.938.0.1 `unordered_set` constructors [unord.set.cnstr]

```
unordered_set() : unordered_set(size_type(see below)) { }
explicit unordered_set(size_type n,
                       const hasher& hf = hasher(),
                       const key_equal& eql = key_equal(),
                       const allocator_type& a = allocator_type());
```

1 *Effects:* Constructs an empty `unordered_set` using the specified hash function, key equality function, and allocator, and using at least *n* buckets. For the default constructor, the number of buckets is implementation-defined. `max_load_factor()` returns 1.0.

2 *Complexity:* Constant.

```
template <class InputIterator>
  unordered_set(InputIterator f, InputIterator l,
                size_type n = see below,
                const hasher& hf = hasher(),
                const key_equal& eql = key_equal(),
                const allocator_type& a = allocator_type());
```

3 *Effects:* Constructs an empty `unordered_set` using the specified hash function, key equality function, and allocator, and using at least *n* buckets. If *n* is not provided, the number of buckets is implementation-defined. Then inserts elements from the range `[`*f*`, `*l*`)`. `max_load_factor()` returns 1.0.

4 *Complexity:* Average case linear, worst case quadratic.

23.938.0.2 `unordered_set` swap [unord.set.swap]

```
template <class Key, class Hash, class Pred, class Alloc>
  void swap(unordered_set<Key, Hash, Pred, Alloc>& x,
            unordered_set<Key, Hash, Pred, Alloc>& y);
```

1 *Effects:* `x.swap(y)`.

23.938.1 Class template `unordered_multiset` [unord.multiset]

23.938.1.1 Class template `unordered_multiset` overview [unord.multiset.overview]

1 An `unordered_multiset` is an unordered associative container that supports equivalent keys (an instance of `unordered_multiset` may contain multiple copies of the same key value) and in which each element's key is the element itself. The `unordered_multiset` class supports forward iterators.

2 An `unordered_multiset` satisfies all of the requirements of a container, of an unordered associative container, and of an allocator-aware container (Table 99). It provides the operations described in the preceding requirements table for equivalent keys; that is, an `unordered_multiset` supports the `a_eq` operations in that table, not the `a_uniq` operations. For an `unordered_multiset<Key>` the `key type` and the value type are both `Key`. The `iterator` and `const_iterator` types are both const iterator types. It is unspecified whether they are the same type.

3 This section only describes operations on `unordered_multiset` that are not described in one of the requirement tables, or for which there is additional semantic information.

```
namespace std {
  template <class Key,
            class Hash  = hash<Key>,
            class Pred  = std::equal_to<Key>,
            class Allocator = std::allocator<Key> >
  class unordered_multiset
```

```
{
public:
  // types
  typedef Key                                                  key_type;
  typedef Key                                                  value_type;
  typedef Hash                                                 hasher;
  typedef Pred                                                 key_equal;
  typedef Allocator                                            allocator_type;
  typedef typename allocator_traits<Allocator>::pointer        pointer;
  typedef typename allocator_traits<Allocator>::const_pointer  const_pointer;
  typedef value_type&                                          reference;
  typedef const value_type&                                    const_reference;
  typedef implementation-defined                               size_type;
  typedef implementation-defined                               difference_type;

  typedef implementation-defined                               iterator;
  typedef implementation-defined                               const_iterator;
  typedef implementation-defined                               local_iterator;
  typedef implementation-defined                               const_local_iterator;

  // construct/destroy/copy
  unordered_multiset();
  explicit unordered_multiset(size_type n,
                              const hasher& hf = hasher(),
                              const key_equal& eql = key_equal(),
                              const allocator_type& a = allocator_type());
  template <class InputIterator>
    unordered_multiset(InputIterator f, InputIterator l,
                       size_type n = see below,
                       const hasher& hf = hasher(),
                       const key_equal& eql = key_equal(),
                       const allocator_type& a = allocator_type());
  unordered_multiset(const unordered_multiset&);
  unordered_multiset(unordered_multiset&&);
  explicit unordered_multiset(const Allocator&);
  unordered_multiset(const unordered_multiset&, const Allocator&);
  unordered_multiset(unordered_multiset&&, const Allocator&);
  unordered_multiset(initializer_list<value_type>,
    size_type = see below,
    const hasher& hf = hasher(),
    const key_equal& eql = key_equal(),
    const allocator_type& a = allocator_type());
  unordered_multiset(size_type n, const allocator_type& a)
    : unordered_multiset(n, hasher(), key_equal(), a) { }
  unordered_multiset(size_type n, const hasher& hf, const allocator_type& a)
    : unordered_multiset(n, hf, key_equal(), a) { }
  template <class InputIterator>
    unordered_multiset(InputIterator f, InputIterator l, size_type n, const allocator_type& a)
    : unordered_multiset(f, l, n, hasher(), key_equal(), a) { }
  template <class InputIterator>
    unordered_multiset(InputIterator f, InputIterator l, size_type n, const hasher& hf,
    const allocator_type& a)
    : unordered_multiset(f, l, n, hf, key_equal(), a) { }
  unordered_multiset(initializer_list<value_type> il, size_type n, const allocator_type& a)
    : unordered_multiset(il, n, hasher(), key_equal(), a) { }
```

```
unordered_multiset(initializer_list<value_type> il, size_type n, const hasher& hf,
const allocator_type& a)
  : unordered_multiset(il, n, hf, key_equal(), a) { }
~unordered_multiset();
unordered_multiset& operator=(const unordered_multiset&);
unordered_multiset& operator=(unordered_multiset&&);
unordered_multiset& operator=(initializer_list<value_type>);
allocator_type get_allocator() const noexcept;

// size and capacity
bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;

// iterators
iterator       begin() noexcept;
const_iterator begin() const noexcept;
iterator       end() noexcept;
const_iterator end() const noexcept;
const_iterator cbegin() const noexcept;
const_iterator cend() const noexcept;

// modifiers
template <class... Args> iterator emplace(Args&&... args);
template <class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
iterator insert(const value_type& obj);
iterator insert(value_type&& obj);
iterator insert(const_iterator hint, const value_type& obj);
iterator insert(const_iterator hint, value_type&& obj);
template <class InputIterator> void insert(InputIterator first, InputIterator last);
void insert(initializer_list<value_type>);

iterator erase(const_iterator position);
size_type erase(const key_type& k);
iterator erase(const_iterator first, const_iterator last);
void clear() noexcept;

void swap(unordered_multiset&);

// observers
hasher hash_function() const;
key_equal key_eq() const;

// lookup
iterator       find(const key_type& k);
const_iterator find(const key_type& k) const;
size_type count(const key_type& k) const;
std::pair<iterator, iterator>             equal_range(const key_type& k);
std::pair<const_iterator, const_iterator> equal_range(const key_type& k) const;

// bucket interface
size_type bucket_count() const noexcept;
size_type max_bucket_count() const noexcept;
size_type bucket_size(size_type n) const;
size_type bucket(const key_type& k) const;
```

```
    local_iterator begin(size_type n);
    const_local_iterator begin(size_type n) const;
    local_iterator end(size_type n);
    const_local_iterator end(size_type n) const;
    const_local_iterator cbegin(size_type n) const;
    const_local_iterator cend(size_type n) const;

    // hash policy
    float load_factor() const noexcept;
    float max_load_factor() const noexcept;
    void max_load_factor(float z);
    void rehash(size_type n);
    void reserve(size_type n);
  };

  template <class Key, class Hash, class Pred, class Alloc>
    void swap(unordered_multiset<Key, Hash, Pred, Alloc>& x,
              unordered_multiset<Key, Hash, Pred, Alloc>& y);
    template <class Key, class Hash, class Pred, class Alloc>
      bool operator==(const unordered_multiset<Key, Hash, Pred, Alloc>& a,
                      const unordered_multiset<Key, Hash, Pred, Alloc>& b);
    template <class Key, class Hash, class Pred, class Alloc>
      bool operator!=(const unordered_multiset<Key, Hash, Pred, Alloc>& a,
                      const unordered_multiset<Key, Hash, Pred, Alloc>& b);
}
```

23.938.1.2 `unordered_multiset` constructors [unord.multiset.cnstr]

```
unordered_multiset() : unordered_multiset(size_type(see below)) { }
explicit unordered_multiset(size_type n,
                            const hasher& hf = hasher(),
                            const key_equal& eql = key_equal(),
                            const allocator_type& a = allocator_type());
```

1 *Effects:* Constructs an empty `unordered_multiset` using the specified hash function, key equality function, and allocator, and using at least *n* buckets. For the default constructor, the number of buckets is implementation-defined. `max_load_factor()` returns 1.0.

2 *Complexity:* Constant.

```
template <class InputIterator>
  unordered_multiset(InputIterator f, InputIterator l,
                     size_type n = see below,
                     const hasher& hf = hasher(),
                     const key_equal& eql = key_equal(),
                     const allocator_type& a = allocator_type());
```

3 *Effects:* Constructs an empty `unordered_multiset` using the specified hash function, key equality function, and allocator, and using at least *n* buckets. If *n* is not provided, the number of buckets is implementation-defined. Then inserts elements from the range `[`*f*`, `*l*`)`. `max_load_factor()` returns 1.0.

4 *Complexity:* Average case linear, worst case quadratic.

23.938.1.3 `unordered_multiset` swap [unord.multiset.swap]

```
template <class Key, class Hash, class Pred, class Alloc>
```

```
void swap(unordered_multiset<Key, Hash, Pred, Alloc>& x,
          unordered_multiset<Key, Hash, Pred, Alloc>& y);
```

1 *Effects:* `x.swap(y);`

23.939 Container adaptors [container.adaptors]

23.939.1 In general [container.adaptors.general]

1 The headers `<queue>` and `<stack>` define the container adaptors `queue`, `priority_queue`, and `stack`.

2 The container adaptors each take a `Container` template parameter, and each constructor takes a `Container` reference argument. This container is copied into the `Container` member of each adaptor. If the container takes an allocator, then a compatible allocator may be passed in to the adaptor's constructor. Otherwise, normal copy or move construction is used for the container argument.

3 For container adaptors, no `swap` function throws an exception unless that exception is thrown by the swap of the adaptor's `Container` or `Compare` object (if any).

23.939.2 Header `<queue>` synopsis [queue.syn]

```
#include <initializer_list>

namespace std {

  template <class T, class Container = deque<T> > class queue;
  template <class T, class Container = vector<T>,
    class Compare = less<typename Container::value_type> >
      class priority_queue;

  template <class T, class Container>
    bool operator==(const queue<T, Container>& x,const queue<T, Container>& y);
  template <class T, class Container>
    bool operator< (const queue<T, Container>& x,const queue<T, Container>& y);
  template <class T, class Container>
    bool operator!=(const queue<T, Container>& x,const queue<T, Container>& y);
  template <class T, class Container>
    bool operator> (const queue<T, Container>& x,const queue<T, Container>& y);
  template <class T, class Container>
    bool operator>=(const queue<T, Container>& x,const queue<T, Container>& y);
  template <class T, class Container>
    bool operator<=(const queue<T, Container>& x,const queue<T, Container>& y);

  template <class T, class Container>
    void swap(queue<T, Container>& x, queue<T, Container>& y) noexcept(noexcept(x.swap(y)));
  template <class T, class Container, class Compare>
    void swap(priority_queue<T, Container, Compare>& x,
              priority_queue<T, Container, Compare>& y) noexcept(noexcept(x.swap(y)));
}
```

23.939.3 Class template `queue` [queue]

23.939.3.1 `queue` definition [queue.defn]

1

Any sequence container supporting operations `front()`, `back()`, `push_back()` and `pop_front()` can be used to instantiate `queue`. In particular, `list` (23.672.1) and `deque` (23.588.1) can be used.

```
namespace std {
```

```
  template <class T, class Container = deque<T> >
  class queue {
  public:
    typedef typename Container::value_type           value_type;
    typedef typename Container::reference            reference;
    typedef typename Container::const_reference      const_reference;
    typedef typename Container::size_type            size_type;
    typedef          Container                       container_type;
  protected:
    Container c;

  public:
    explicit queue(const Container&);
    explicit queue(Container&& = Container());
    template <class Alloc> explicit queue(const Alloc&);
    template <class Alloc> queue(const Container&, const Alloc&);
    template <class Alloc> queue(Container&&, const Alloc&);
    template <class Alloc> queue(const queue&, const Alloc&);
    template <class Alloc> queue(queue&&, const Alloc&);

    bool              empty() const     { return c.empty(); }
    size_type         size()  const     { return c.size(); }
    reference         front()           { return c.front(); }
    const_reference   front() const     { return c.front(); }
    reference         back()            { return c.back(); }
    const_reference   back() const      { return c.back(); }
    void push(const value_type& x)      { c.push_back(x); }
    void push(value_type&& x)           { c.push_back(std::move(x)); }
    template <class... Args> void emplace(Args&&... args)
      { c.emplace_back(std::forward<Args>(args)...); }
    void pop()                          { c.pop_front(); }
    void swap(queue& q) noexcept(noexcept(swap(c, q.c)))
      { using std::swap; swap(c, q.c); }
  };

  template <class T, class Container>
    bool operator==(const queue<T, Container>& x, const queue<T, Container>& y);
  template <class T, class Container>
    bool operator< (const queue<T, Container>& x, const queue<T, Container>& y);
  template <class T, class Container>
    bool operator!=(const queue<T, Container>& x, const queue<T, Container>& y);
  template <class T, class Container>
    bool operator> (const queue<T, Container>& x, const queue<T, Container>& y);
  template <class T, class Container>
    bool operator>=(const queue<T, Container>& x, const queue<T, Container>& y);
  template <class T, class Container>
    bool operator<=(const queue<T, Container>& x, const queue<T, Container>& y);

  template <class T, class Container>
    void swap(queue<T, Container>& x, queue<T, Container>& y) noexcept(noexcept(x.swap(y)));

  template <class T, class Container, class Alloc>
    struct uses_allocator<queue<T, Container>, Alloc>
      : uses_allocator<Container, Alloc>::type { };
}
```

23.941.0.1 queue constructors [queue.cons]

```
explicit queue(const Container& cont);
```

1 *Effects:* Initializes `c` with `cont`.

```
explicit queue(Container&& cont = Container());
```

2 *Effects:* Initializes `c` with `std::move(cont)`.

23.941.0.2 queue constructors with allocators [queue.cons.alloc]

1 If `uses_allocator<container_type, Alloc>::value` is `false` the constructors in this subclause shall not participate in overload resolution.

```
template <class Alloc>
  explicit queue(const Alloc& a);
```

2 *Effects:* Initializes `c` with `a`.

```
template <class Alloc>
  queue(const container_type& cont, const Alloc& a);
```

3 *Effects:* Initializes `c` with `cont` as the first argument and `a` as the second argument.

```
template <class Alloc>
  queue(container_type&& cont, const Alloc& a);
```

4 *Effects:* Initializes `c` with `std::move(cont)` as the first argument and `a` as the second argument.

```
template <class Alloc>
  queue(const queue& q, const Alloc& a);
```

5 *Effects:* Initializes `c` with `q.c` as the first argument and `a` as the second argument.

```
template <class Alloc>
  queue(queue&& q, const Alloc& a);
```

6 *Effects:* Initializes `c` with `std::move(q.c)` as the first argument and `a` as the second argument.

23.941.0.3 queue operators [queue.ops]

```
template <class T, class Container>
    bool operator==(const queue<T, Container>& x,
                    const queue<T, Container>& y);
```

1 *Returns:* `x.c == y.c`.

```
template <class T, class Container>
    bool operator!=(const queue<T, Container>& x,
                    const queue<T, Container>& y);
```

2 *Returns:* `x.c != y.c`.

```
template <class T, class Container>
    bool operator< (const queue<T, Container>& x,
                    const queue<T, Container>& y);
```

3 *Returns:* `x.c < y.c`.

```
template <class T, class Container>
    bool operator<=(const queue<T, Container>& x,
                    const queue<T, Container>& y);
```

4 *Returns:* `x.c <= y.c`.

```
template <class T, class Container>
    bool operator> (const queue<T, Container>& x,
                    const queue<T, Container>& y);
```

5 *Returns:* `x.c > y.c`.

```
template <class T, class Container>
    bool operator>=(const queue<T, Container>& x,
                    const queue<T, Container>& y);
```

6 *Returns:* `x.c >= y.c`.

23.941.0.4 queue specialized algorithms [queue.special]

```
template <class T, class Container>
  void swap(queue<T, Container>& x, queue<T, Container>& y) noexcept(noexcept(x.swap(y)));
```

1 *Effects:* `x.swap(y)`.

23.941.1 Class template `priority_queue` [priority.queue]

1

Any sequence container with random access iterator and supporting operations `front()`, `push_back()` and `pop_back()` can be used to instantiate `priority_queue`. In particular, `vector` (23.696.1) and `deque` (23.588.1) can be used. Instantiating `priority_queue` also involves supplying a function or function object for making priority comparisons; the library assumes that the function or function object defines a strict weak ordering (25.98).

```
namespace std {
  template <class T, class Container = vector<T>,
    class Compare = less<typename Container::value_type> >
  class priority_queue {
  public:
    typedef typename Container::value_type            value_type;
    typedef typename Container::reference             reference;
    typedef typename Container::const_reference       const_reference;
    typedef typename Container::size_type             size_type;
    typedef          Container                        container_type;
  protected:
    Container c;
    Compare comp;

  public:
    priority_queue(const Compare& x, const Container&);
    explicit priority_queue(const Compare& x = Compare(), Container&& = Container());
    template <class InputIterator>
      priority_queue(InputIterator first, InputIterator last,
             const Compare& x, const Container&);
    template <class InputIterator>
      priority_queue(InputIterator first, InputIterator last,
             const Compare& x = Compare(), Container&& = Container());
    template <class Alloc> explicit priority_queue(const Alloc&);
    template <class Alloc> priority_queue(const Compare&, const Alloc&);
    template <class Alloc> priority_queue(const Compare&,
      const Container&, const Alloc&);
```

```
    template <class Alloc> priority_queue(const Compare&,
      Container&&, const Alloc&);
    template <class Alloc> priority_queue(const priority_queue&, const Alloc&);
    template <class Alloc> priority_queue(priority_queue&&, const Alloc&);

    bool      empty() const       { return c.empty(); }
    size_type size()  const       { return c.size(); }
    const_reference   top() const { return c.front(); }
    void push(const value_type& x);
    void push(value_type&& x);
    template <class... Args> void emplace(Args&&... args);
    void pop();
    void swap(priority_queue& q) noexcept(
        noexcept(swap(c, q.c)) && noexcept(swap(comp, q.comp)))
      { using std::swap; swap(c, q.c); swap(comp, q.comp); }
  };
  // no equality is provided
  template <class T, class Container, class Compare>
    void swap(priority_queue<T, Container, Compare>& x,
              priority_queue<T, Container, Compare>& y) noexcept(noexcept(x.swap(y)));

  template <class T, class Container, class Compare, class Alloc>
    struct uses_allocator<priority_queue<T, Container, Compare>, Alloc>
      : uses_allocator<Container, Alloc>::type { };
}
```

23.943.0.1 `priority_queue` constructors [priqueue.cons]

```
priority_queue(const Compare& x,
               const Container& y);
explicit priority_queue(const Compare& x = Compare(),
               Container&& y = Container());
```

1 *Requires:* `x` shall define a strict weak ordering (25.98).

2 *Effects:* Initializes `comp` with `x` and `c` with `y` (copy constructing or move constructing as appropriate); calls `make_heap(c.begin(), c.end(), comp)`.

```
template <class InputIterator>
  priority_queue(InputIterator first, InputIterator last,
                 const Compare& x,
                 const Container& y);
template <class InputIterator>
  priority_queue(InputIterator first, InputIterator last,
                 const Compare& x = Compare(),
                 Container&& y = Container());
```

1 *Requires:* `x` shall define a strict weak ordering (25.98).

2 *Effects:* Initializes `comp` with `x` and `c` with `y` (copy constructing or move constructing as appropriate); calls `c.insert(c.end(), first, last)`; and finally calls `make_heap(c.begin(), c.end(), comp)`.

23.945.0.1 `priority_queue` constructors with allocators [priqueue.cons.alloc]

1 If `uses_allocator<container_type, Alloc>::value` is `false` the constructors in this subclause shall not participate in overload resolution.

```
template <class Alloc>
  explicit priority_queue(const Alloc& a);
```

2 *Effects:* Initializes `c` with `a` and value-initializes `comp`.

```
template <class Alloc>
  priority_queue(const Compare& compare, const Alloc& a);
```

3 *Effects:* Initializes `c` with `a` and initializes `comp` with `compare`.

```
template <class Alloc>
  priority_queue(const Compare& compare, const Container& cont, const Alloc& a);
```

1 *Effects:* Initializes `c` with `cont` as the first argument and `a` as the second argument, and initializes `comp` with `compare`.

```
template <class Alloc>
  priority_queue(const Compare& compare, Container&& cont, const Alloc& a);
```

1 *Effects:* Initializes `c` with `std::move(cont)` as the first argument and `a` as the second argument, and initializes `comp` with `compare`.

```
template <class Alloc>
  priority_queue(const priority_queue& q, const Alloc& a);
```

2 *Effects:* Initializes `c` with `q.c` as the first argument and `a` as the second argument, and initializes `comp` with `q.comp`.

```
template <class Alloc>
  priority_queue(priority_queue&& q, const Alloc& a);
```

3 *Effects:* Initializes `c` with `std::move(q.c)` as the first argument and `a` as the second argument, and initializes `comp` with `std::move(q.comp)`.

23.947.0.1 `priority_queue` members [priqueue.members]

```
void push(const value_type& x);
```

1 *Effects:*

```
c.push_back(x);
push_heap(c.begin(), c.end(), comp);
```

```
void push(value_type&& x);
```

2 *Effects:*

```
c.push_back(std::move(x));
push_heap(c.begin(), c.end(), comp);
```

```
template <class... Args> void emplace(Args&&... args)
```

3 *Effects:*

```
c.emplace_back(std::forward<Args>(args)...);
push_heap(c.begin(), c.end(), comp);
```

```
void pop();
```

4 *Effects:*

```
pop_heap(c.begin(), c.end(), comp);
c.pop_back();
```

23.947.0.2 priority_queue specialized algorithms [priqueue.special]

```
template <class T, class Container, Compare>
  void swap(priority_queue<T, Container, Compare>& x,
            priority_queue<T, Container, Compare>& y) noexcept(noexcept(x.swap(y)));
```

1 *Effects:* `x.swap(y)`.

23.947.1 Class template stack [stack]

1 Any sequence container supporting operations `back()`, `push_back()` and `pop_back()` can be used to instantiate `stack`. In particular, `vector` (23.696.1), `list` (23.672.1) and `deque` (23.588.1) can be used.

23.947.1.1 Header <stack> synopsis [stack.syn]

```
#include <initializer_list>

namespace std {

  template <class T, class Container = deque<T> > class stack;
  template <class T, class Container>
    bool operator==(const stack<T, Container>& x,const stack<T, Container>& y);
  template <class T, class Container>
    bool operator< (const stack<T, Container>& x,const stack<T, Container>& y);
  template <class T, class Container>
    bool operator!=(const stack<T, Container>& x,const stack<T, Container>& y);
  template <class T, class Container>
    bool operator> (const stack<T, Container>& x,const stack<T, Container>& y);
  template <class T, class Container>
    bool operator>=(const stack<T, Container>& x,const stack<T, Container>& y);
  template <class T, class Container>
    bool operator<=(const stack<T, Container>& x,const stack<T, Container>& y);
  template <class T, class Container>
    void swap(stack<T, Container>& x, stack<T, Container>& y) noexcept(noexcept(x.swap(y)));
}
```

23.947.1.2 stack definition [stack.defn]

```
namespace std {
  template <class T, class Container = deque<T> >
  class stack {
  public:
    typedef typename Container::value_type            value_type;
    typedef typename Container::reference             reference;
    typedef typename Container::const_reference       const_reference;
    typedef typename Container::size_type             size_type;
    typedef          Container                        container_type;
  protected:
    Container c;

  public:
```

```
    explicit stack(const Container&);
    explicit stack(Container&& = Container());
    template <class Alloc> explicit stack(const Alloc&);
    template <class Alloc> stack(const Container&, const Alloc&);
    template <class Alloc> stack(Container&&, const Alloc&);
    template <class Alloc> stack(const stack&, const Alloc&);
    template <class Alloc> stack(stack&&, const Alloc&);

    bool      empty() const            { return c.empty(); }
    size_type size()  const            { return c.size(); }
    reference         top()            { return c.back(); }
    const_reference   top() const      { return c.back(); }
    void push(const value_type& x)     { c.push_back(x); }
    void push(value_type&& x)          { c.push_back(std::move(x)); }
    template <class... Args> void emplace(Args&&... args)
      { c.emplace_back(std::forward<Args>(args)...); }
    void pop()                         { c.pop_back(); }
    void swap(stack& s) noexcept(noexcept(swap(c, s.c)))
      { using std::swap; swap(c, s.c); }
  };

  template <class T, class Container>
    bool operator==(const stack<T, Container>& x, const stack<T, Container>& y);
  template <class T, class Container>
    bool operator< (const stack<T, Container>& x, const stack<T, Container>& y);
  template <class T, class Container>
    bool operator!=(const stack<T, Container>& x, const stack<T, Container>& y);
  template <class T, class Container>
    bool operator> (const stack<T, Container>& x, const stack<T, Container>& y);
  template <class T, class Container>
    bool operator>=(const stack<T, Container>& x, const stack<T, Container>& y);
  template <class T, class Container>
    bool operator<=(const stack<T, Container>& x, const stack<T, Container>& y);
  template <class T, class Container>
    void swap(stack<T, Container>& x, stack<T, Container>& y) noexcept(noexcept(x.swap(y)));

  template <class T, class Container, class Alloc>
    struct uses_allocator<stack<T, Container>, Alloc>
      : uses_allocator<Container, Alloc>::type { };
}
```

23.947.1.3 stack constructors [stack.cons]

`explicit stack(const Container& cont);`

1 *Effects:* Initializes `c` with `cont`.

`explicit stack(Container&& cont = Container());`

2 *Effects:* Initializes `c` with `std::move(cont)`.

23.947.1.4 stack constructors with allocators [stack.cons.alloc]

1 If `uses_allocator<container_type, Alloc>::value` is `false` the constructors in this subclause shall not participate in overload resolution.

```
template <class Alloc>
  explicit stack(const Alloc& a);
```

2 *Effects:* Initializes `c` with `a`.

```
template <class Alloc>
  stack(const container_type& cont, const Alloc& a);
```

3 *Effects:* Initializes `c` with `cont` as the first argument and `a` as the second argument.

```
template <class Alloc>
  stack(container_type&& cont, const Alloc& a);
```

4 *Effects:* Initializes `c` with `std::move(cont)` as the first argument and `a` as the second argument.

```
template <class Alloc>
  stack(const stack& s, const Alloc& a);
```

5 *Effects:* Initializes `c` with `s.c` as the first argument and `a` as the second argument.

```
template <class Alloc>
  stack(stack&& s, const Alloc& a);
```

6 *Effects:* Initializes `c` with `std::move(s.c)` as the first argument and `a` as the second argument.

23.947.1.5 `stack` operators [stack.ops]

```
template <class T, class Container>
    bool operator==(const stack<T, Container>& x,
                    const stack<T, Container>& y);
```

1 *Returns:* `x.c == y.c`.

```
template <class T, class Container>
    bool operator!=(const stack<T, Container>& x,
                    const stack<T, Container>& y);
```

2 *Returns:* `x.c != y.c`.

```
template <class T, class Container>
    bool operator< (const stack<T, Container>& x,
                    const stack<T, Container>& y);
```

3 *Returns:* `x.c < y.c`.

```
template <class T, class Container>
    bool operator<=(const stack<T, Container>& x,
                    const stack<T, Container>& y);
```

4 *Returns:* `x.c <= y.c`.

```
template <class T, class Container>
    bool operator> (const stack<T, Container>& x,
                    const stack<T, Container>& y);
```

5 *Returns:* `x.c > y.c`.

```
template <class T, class Container>
    bool operator>=(const stack<T, Container>& x,
                    const stack<T, Container>& y);
```

6 *Returns:* `x.c >= y.c`.

23.947.1.6 stack specialized algorithms [stack.special]

```
template <class T, class Container>
  void swap(stack<T, Container>& x, stack<T, Container>& y) noexcept(noexcept(x.swap(y)));
```

1 *Effects:* `x.swap(y)`.

24 Iterators library [iterators]

24.1 General [iterators.general]

1 This Clause describes components that C++ programs may use to perform iterations over containers (Clause 23), streams (27.109), and stream buffers (27.92).

2 The following subclauses describe iterator requirements, and components for iterator primitives, predefined iterators, and stream iterators, as summarized in Table 104.

Table 104 — Iterators library summary

	Subclause	Header(s)
24.2	Requirements	
24.172	Iterator primitives	`<iterator>`
24.197	Predefined iterators	
24.207	Stream iterators	

24.2 Iterator requirements [iterator.requirements]

24.2.1 In general [iterator.requirements.general]

1

Iterators are a generalization of pointers that allow a C++ program to work with different data structures (containers) in a uniform manner. To be able to construct template algorithms that work correctly and efficiently on different types of data structures, the library formalizes not just the interfaces but also the semantics and complexity assumptions of iterators. All input iterators `i` support the expression `*i`, resulting in a value of some object type `T`, called the *value type* of the iterator. All output iterators support the expression `*i = o` where `o` is a value of some type that is in the set of types that are *writable* to the particular iterator type of `i`. All iterators `i` for which the expression `(*i).m` is well-defined, support the expression `i->m` with the same semantics as `(*i).m`. For every iterator type `X` for which equality is defined, there is a corresponding signed integer type called the *difference type* of the iterator.

1

Since iterators are an abstraction of pointers, their semantics is a generalization of most of the semantics of pointers in C++. This ensures that every function template that takes iterators works as well with regular pointers. This International Standard defines five categories of iterators, according to the operations defined on them: *input iterators*, *output iterators*, *forward iterators*, *bidirectional iterators* and *random access iterators*, as shown in Table 105.

Table 105 — Relations among iterator categories

Random Access	→ **Bidirectional**	→ **Forward**	→ **Input**
			→ **Output**

1

Forward iterators satisfy all the requirements of input iterators and can be used whenever an input iterator is specified; Bidirectional iterators also satisfy all the requirements of forward iterators and can be used whenever a forward iterator is specified; Random access iterators also satisfy all the requirements of bidirectional iterators and can be used whenever a bidirectional iterator is specified.

1

Iterators that further satisfy the requirements of output iterators are called *mutable iterator*s. Nonmutable iterators are referred to as *constant iterator*s.

1

Just as a regular pointer to an array guarantees that there is a pointer value pointing past the last element of the array, so for any iterator type there is an iterator value that points past the last element of a corresponding sequence. These values are called *past-the-end* values. Values of an iterator `i` for which the expression `*i` is defined are called *dereferenceable*. The library never assumes that past-the-end values are dereferenceable. Iterators can also have singular values that are not associated with any sequence. [*Example:* After the declaration of an uninitialized pointer `x` (as with `int* x;`), `x` must always be assumed to have a singular value of a pointer. — *end example*] Results of most expressions are undefined for singular values; the only exceptions are destroying an iterator that holds a singular value, the assignment of a non-singular value to an iterator that holds a singular value, and, for iterators that satisfy the `DefaultConstructible` requirements, using a value-initialized iterator as the source of a copy or move operation. [*Note:* This guarantee is not offered for default initialization, although the distinction only matters for types with trivial default constructors such as pointers or aggregates holding pointers. — *end note*] In these cases the singular value is overwritten the same way as any other value. Dereferenceable values are always non-singular.

1

An iterator `j` is called *reachable* from an iterator `i` if and only if there is a finite sequence of applications of the expression `++i` that makes `i == j`. If `j` is reachable from `i`, they refer to elements of the same sequence.

1

Most of the library's algorithmic templates that operate on data structures have interfaces that use ranges. A *range* is a pair of iterators that designate the beginning and end of the computation. A range `[i,i)` is an empty range; in general, a range `[i,j)` refers to the elements in the data structure starting with the element pointed to by `i` and up to but not including the element pointed to by `j`. Range `[i,j)` is valid if and only if `j` is reachable from `i`. The result of the application of functions in the library to invalid ranges is undefined.

1

All the categories of iterators require only those functions that are realizable for a given category in constant time (amortized). Therefore, requirement tables for the iterators do not have a complexity column.

1

Destruction of an iterator may invalidate pointers and references previously obtained from that iterator.

1

An *invalid* iterator is an iterator that may be singular.[267]

1

In the following sections, `a` and `b` denote values of type `X` or `const X`, `difference_type` and `reference` refer to the types `iterator_traits<X>::difference_type` and `iterator_traits<X>::reference`, respectively, `n` denotes a value of `difference_type`, `u`, `tmp`, and `m` denote identifiers, `r` denotes a value of `X&`, `t` denotes a value of value type `T`, `o` denotes a value of some type that is writable to the output iterator. [*Note:* For an iterator type `X` there must be an instantiation of `iterator_traits<X>` (24.172.1). — *end note*]

24.95.1 Iterator [iterator.iterators]

1

267) This definition applies to pointers, since pointers are iterators. The effect of dereferencing an iterator that has been invalidated is undefined.

The `Iterator` requirements form the basis of the iterator concept taxonomy; every iterator satisfies the `Iterator` requirements. This set of requirements specifies operations for dereferencing and incrementing an iterator. Most algorithms will require additional operations to read (24.100.1) or write (24.116.1) values, or to provide a richer set of iterator movements (24.122.1, 24.165.1, 24.167.1).)

1

A type `X` satisfies the `Iterator` requirements if:

— `X` satisfies the `CopyConstructible`, `CopyAssignable`, and `Destructible` requirements (17.128.1.1) and lvalues of type `X` are swappable (17.143.0.1), and

— the expressions in Table 106 are valid and have the indicated semantics.

Table 106 — Iterator requirements

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
`*r`	`reference`		pre: `r` is dereferenceable.
`++r`	`X&`		

24.100.1 Input iterators [input.iterators]

1

A class or pointer type `X` satisfies the requirements of an input iterator for the value type `T` if X satisfies the `Iterator` (24.95.1) and `EqualityComparable` (Table 17) requirements and the expressions in Table 107 are valid and have the indicated semantics.

1

In Table 107, the term *the domain of* `==` is used in the ordinary mathematical sense to denote the set of values over which `==` is (required to be) defined. This set can change over time. Each algorithm places additional requirements on the domain of `==` for the iterator values it uses. These requirements can be inferred from the uses that algorithm makes of `==` and `!=`. [*Example:* the call `find(a,b,x)` is defined only if the value of `a` has the property p defined as follows: `b` has property p and a value `i` has property p if `(*i==x)` or if `(*i!=x` and `++i` has property `p`). — *end example*]

Table 107 — Input iterator requirements (in addition to Iterator)

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
`a != b`	contextually convertible to `bool`	`!(a == b)`	pre: `(a,b)` is in the domain of `==`.
`*a`	convertible to `T`		pre: `a` is dereferenceable. The expression `(void)*a, *a` is equivalent to `*a`. If `a == b` and `(a,b)` is in the domain of `==` then `*a` is equivalent to `*b`.
`a->m`		`(*a).m`	pre: `a` is dereferenceable.

Table 107 — Input iterator requirements (in addition to Iterator) (continued)

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
`++r`	`X&`		pre: `r` is dereferenceable. post: `r` is dereferenceable or `r` is past-the-end. post: any copies of the previous value of `r` are no longer required either to be dereferenceable or to be in the domain of `==`.
`(void)r++`			equivalent to `(void)++r`
`*r++`	convertible to `T`	`{ T tmp = *r; ++r; return tmp; }`	

1

[*Note:* For input iterators, `a == b` does not imply `++a == ++b`. (Equality does not guarantee the substitution property or referential transparency.) Algorithms on input iterators should never attempt to pass through the same iterator twice. They should be *single pass* algorithms. Value type T is not required to be a `CopyAssignable` type (Table 23). These algorithms can be used with istreams as the source of the input data through the `istream_iterator` class template. — *end note*]

24.116.1 Output iterators [output.iterators]

1

A class or pointer type `X` satisfies the requirements of an output iterator if `X` satisfies the `Iterator` requirements (24.95.1) and the expressions in Table 108 are valid and have the indicated semantics.

Table 108 — Output iterator requirements (in addition to Iterator)

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
`*r = o`	result is not used		*Remark:* After this operation `r` is not required to be dereferenceable. post: `r` is incrementable.
`++r`	`X&`		`&r == &++r`. *Remark:* After this operation `r` is not required to be dereferenceable. post: `r` is incrementable.
`r++`	convertible to `const X&`	`{ X tmp = r; ++r; return tmp; }`	*Remark:* After this operation `r` is not required to be dereferenceable. post: `r` is incrementable.

Table 108 — Output iterator requirements (in addition to Iterator) (continued)

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
`*r++ = o`	result is not used		*Remark:* After this operation `r` is not required to be dereferenceable. post: `r` is incrementable.

1

[*Note:* The only valid use of an `operator*` is on the left side of the assignment statement. *Assignment through the same value of the iterator happens only once.* Algorithms on output iterators should never attempt to pass through the same iterator twice. They should be *single pass* algorithms. Equality and inequality might not be defined. Algorithms that take output iterators can be used with ostreams as the destination for placing data through the `ostream_iterator` class as well as with insert iterators and insert pointers. — *end note*]

24.122.1 Forward iterators [forward.iterators]

1

A class or pointer type `X` satisfies the requirements of a forward iterator if

— `X` satisfies the requirements of an input iterator (24.100.1),

— X satisfies the `DefaultConstructible` requirements (17.128.1.1),

— if `X` is a mutable iterator, `reference` is a reference to `T`; if `X` is a const iterator, `reference` is a reference to `const T`,

— the expressions in Table 109 are valid and have the indicated semantics, and

— objects of type `X` offer the multi-pass guarantee, described below.

1

The domain of == for forward iterators is that of iterators over the same underlying sequence. However, value-initialized iterators may be compared and shall compare equal to other value-initialized iterators of the same type. [*Note:* value initialized iterators behave as if they refer past the end of the same empty sequence — *end note*]

1

Two dereferenceable iterators `a` and `b` of type `X` offer the *multi-pass guarantee* if:

— `a == b` implies `++a == ++b` and

— `X` is a pointer type or the expression `(void)++X(a), *a` is equivalent to the expression `*a`.

1

[*Note:* The requirement that `a == b` implies `++a == ++b` (which is not true for input and output iterators) and the removal of the restrictions on the number of the assignments through a mutable iterator (which applies to output iterators) allows the use of multi-pass one-directional algorithms with forward iterators. — *end note*]

Table 109 — Forward iterator requirements (in addition to input iterator)

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition

`r++`	convertible to `const X&`	`{ X tmp = r;`
		`++r;`
		`return tmp; }`	

`*r++`	`reference`		

1

If `a` and `b` are equal, then either `a` and `b` are both dereferenceable or else neither is dereferenceable.

1

If `a` and `b` are both dereferenceable, then `a == b` if and only if `*a` and `*b` are bound to the same object.

24.165.1 Bidirectional iterators [bidirectional.iterators]

1

A class or pointer type `X` satisfies the requirements of a bidirectional iterator if, in addition to satisfying the requirements for forward iterators, the following expressions are valid as shown in Table 110.

Table 110 — Bidirectional iterator requirements (in addition to forward iterator)

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
`--r`	`X&`		pre: there exists `s` such that `r == ++s`. post: `r` is dereferenceable. `--(++r) == r`. `--r == --s` implies `r == s`. `&r == &--r`.

`r--`	convertible to `const X&`	`{ X tmp = r;`
		`--r;`
		`return tmp; }`	

`*r--`	`reference`		

1 [*Note:* Bidirectional iterators allow algorithms to move iterators backward as well as forward. — *end note*]

24.167.1 Random access iterators [random.access.iterators]

1

A class or pointer type `X` satisfies the requirements of a random access iterator if, in addition to satisfying the requirements for bidirectional iterators, the following expressions are valid as shown in Table 111.

Table 111 — Random access iterator requirements (in addition to bidirectional iterator)

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition

`r += n`	`X&`	`{ difference_type m = n;`
		`if (m >= 0)`
		`while (m--)`
		`++r;`
		`else`
		`while (m++)`
		`--r;`
		`return r; }`	

`a + n` `n + a`	`X`	`{ X tmp = a;` `return tmp += n; }`	`a + n == n + a`.
`r -= n`	`X&`	`return r += -n;`	
`a - n`	`X`	`{ X tmp = a;` `return tmp -= n; }`	
`b - a`	`difference_type`	`return n`	pre: there exists a value `n` of type `difference_type` such that `a + n == b`. `b == a + (b - a)`.
`a[n]`	convertible to `reference`	`*(a + n)`	
`a < b`	contextually convertible to `bool`	`b - a > 0`	`<` is a total ordering relation
`a > b`	contextually convertible to `bool`	`b < a`	`>` is a total ordering relation opposite to `<`.
`a >= b`	contextually convertible to `bool`	`!(a < b)`	
`a <= b`	contextually convertible to `bool`.	`!(a > b)`	

24.171 Header `<iterator>` synopsis [iterator.synopsis]

```
namespace std {
  // 24.172, primitives:
  template<class Iterator> struct iterator_traits;
  template<class T> struct iterator_traits<T*>;

  template<class Category, class T, class Distance = ptrdiff_t,
       class Pointer = T*, class Reference = T&> struct iterator;

  struct input_iterator_tag { };
  struct output_iterator_tag { };
  struct forward_iterator_tag: public input_iterator_tag { };
  struct bidirectional_iterator_tag: public forward_iterator_tag { };
  struct random_access_iterator_tag: public bidirectional_iterator_tag { };
```

```
// 24.185.1, iterator operations:
template <class InputIterator, class Distance>
  void advance(InputIterator& i, Distance n);
template <class InputIterator>
  typename iterator_traits<InputIterator>::difference_type
  distance(InputIterator first, InputIterator last);
template <class ForwardIterator>
  ForwardIterator next(ForwardIterator x,
    typename std::iterator_traits<ForwardIterator>::difference_type n = 1);
template <class BidirectionalIterator>
  BidirectionalIterator prev(BidirectionalIterator x,
    typename std::iterator_traits<BidirectionalIterator>::difference_type n = 1);

// 24.197, predefined iterators:
template <class Iterator> class reverse_iterator;

template <class Iterator1, class Iterator2>
  bool operator==(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
template <class Iterator1, class Iterator2>
  bool operator<(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
template <class Iterator1, class Iterator2>
  bool operator!=(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
template <class Iterator1, class Iterator2>
  bool operator>(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
template <class Iterator1, class Iterator2>
  bool operator>=(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
template <class Iterator1, class Iterator2>
  bool operator<=(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);

template <class Iterator1, class Iterator2>
  auto operator-(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y) ->decltype(y.base() - x.base());
template <class Iterator>
  reverse_iterator<Iterator>
    operator+(
  typename reverse_iterator<Iterator>::difference_type n,
  const reverse_iterator<Iterator>& x);

template <class Iterator>
  reverse_iterator<Iterator> make_reverse_iterator(Iterator i);
```

```
template <class Container> class back_insert_iterator;
template <class Container>
  back_insert_iterator<Container> back_inserter(Container& x);

template <class Container> class front_insert_iterator;
template <class Container>
  front_insert_iterator<Container> front_inserter(Container& x);

template <class Container> class insert_iterator;
template <class Container>
  insert_iterator<Container> inserter(Container& x, typename Container::iterator i);

template <class Iterator> class move_iterator;
template <class Iterator1, class Iterator2>
  bool operator==(
    const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
template <class Iterator1, class Iterator2>
  bool operator!=(
    const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
template <class Iterator1, class Iterator2>
  bool operator<(
    const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
template <class Iterator1, class Iterator2>
  bool operator<=(
    const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
template <class Iterator1, class Iterator2>
  bool operator>(
    const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
template <class Iterator1, class Iterator2>
  bool operator>=(
    const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);

template <class Iterator1, class Iterator2>
  auto operator-(
  const move_iterator<Iterator1>& x,
  const move_iterator<Iterator2>& y) -> decltype(x.base() - y.base());
template <class Iterator>
  move_iterator<Iterator> operator+(
    typename move_iterator<Iterator>::difference_type n, const move_iterator<Iterator>& x);
template <class Iterator>
  move_iterator<Iterator> make_move_iterator(Iterator i);

// 24.207, stream iterators:
template <class T, class charT = char, class traits = char_traits<charT>,
    class Distance = ptrdiff_t>
class istream_iterator;
template <class T, class charT, class traits, class Distance>
  bool operator==(const istream_iterator<T,charT,traits,Distance>& x,
          const istream_iterator<T,charT,traits,Distance>& y);
template <class T, class charT, class traits, class Distance>
  bool operator!=(const istream_iterator<T,charT,traits,Distance>& x,
          const istream_iterator<T,charT,traits,Distance>& y);

template <class T, class charT = char, class traits = char_traits<charT> >
    class ostream_iterator;
```

```
  template<class charT, class traits = char_traits<charT> >
    class istreambuf_iterator;
  template <class charT, class traits>
    bool operator==(const istreambuf_iterator<charT,traits>& a,
            const istreambuf_iterator<charT,traits>& b);
  template <class charT, class traits>
    bool operator!=(const istreambuf_iterator<charT,traits>& a,
            const istreambuf_iterator<charT,traits>& b);

  template <class charT, class traits = char_traits<charT> >
    class ostreambuf_iterator;

  // 24.221, range access:
  template <class C> auto begin(C& c) -> decltype(c.begin());
  template <class C> auto begin(const C& c) -> decltype(c.begin());
  template <class C> auto end(C& c) -> decltype(c.end());
  template <class C> auto end(const C& c) -> decltype(c.end());
  template <class T, size_t N> constexpr T* begin(T (&array)[N]) noexcept;
  template <class T, size_t N> constexpr T* end(T (&array)[N]) noexcept;
  template <class C> constexpr auto cbegin(const C& c) noexcept(noexcept(std::begin(c)))
    -> decltype(std::begin(c));
  template <class C> constexpr auto cend(const C& c) noexcept(noexcept(std::end(c)))
    -> decltype(std::end(c));
  template <class C> auto rbegin(C& c) -> decltype(c.rbegin());
  template <class C> auto rbegin(const C& c) -> decltype(c.rbegin());
  template <class C> auto rend(C& c) -> decltype(c.rend());
  template <class C> auto rend(const C& c) -> decltype(c.rend());
  template <class T, size_t N> reverse_iterator<T*> rbegin(T (&array)[N]);
  template <class T, size_t N> reverse_iterator<T*> rend(T (&array)[N]);
  template <class E> reverse_iterator<const E*> rbegin(initializer_list<E> il);
  template <class E> reverse_iterator<const E*> rend(initializer_list<E> il);
  template <class C> auto crbegin(const C& c) -> decltype(std::rbegin(c));
  template <class C> auto crend(const C& c) -> decltype(std::rend(c));
}
```

24.172 Iterator primitives [iterator.primitives]

1 To simplify the task of defining iterators, the library provides several classes and functions:

24.172.1 Iterator traits [iterator.traits]

1

To implement algorithms only in terms of iterators, it is often necessary to determine the value and difference types that correspond to a particular iterator type. Accordingly, it is required that if `Iterator` is the type of an iterator, the types

```
iterator_traits<Iterator>::difference_type
iterator_traits<Iterator>::value_type
iterator_traits<Iterator>::iterator_category
```

be defined as the iterator's difference type, value type and iterator category, respectively. In addition, the types

```
iterator_traits<Iterator>::reference
iterator_traits<Iterator>::pointer
```

shall be defined as the iterator's reference and pointer types, that is, for an iterator object `a`, the same type as the type of `*a` and `a->`, respectively. In the case of an output iterator, the types

```
iterator_traits<Iterator>::difference_type
iterator_traits<Iterator>::value_type
iterator_traits<Iterator>::reference
iterator_traits<Iterator>::pointer
```

may be defined as `void`.

1 The template `iterator_traits<Iterator>` is defined as

```
namespace std {
  template<class Iterator> struct iterator_traits {
    typedef typename Iterator::difference_type difference_type;
    typedef typename Iterator::value_type value_type;
    typedef typename Iterator::pointer pointer;
    typedef typename Iterator::reference reference;
    typedef typename Iterator::iterator_category iterator_category;
  };
}
```

2 It is specialized for pointers as

```
namespace std {
  template<class T> struct iterator_traits<T*> {
    typedef ptrdiff_t difference_type;
    typedef T value_type;
    typedef T* pointer;
    typedef T& reference;
    typedef random_access_iterator_tag iterator_category;
  };
}
```

and for pointers to const as

```
namespace std {
  template<class T> struct iterator_traits<const T*> {
    typedef ptrdiff_t difference_type;
    typedef T value_type;
    typedef const T* pointer;
    typedef const T& reference;
    typedef random_access_iterator_tag iterator_category;
  };
}
```

3 [*Example:* To implement a generic `reverse` function, a C++ program can do the following:

```
template <class BidirectionalIterator>
void reverse(BidirectionalIterator first, BidirectionalIterator last) {
  typename iterator_traits<BidirectionalIterator>::difference_type n =
    distance(first, last);
  --n;
  while(n > 0) {
    typename iterator_traits<BidirectionalIterator>::value_type
     tmp = *first;
    *first++ = *--last;
    *last = tmp;
```

```
    n -= 2;
  }
}
```

— *end example*]

24.182.1 Basic iterator [iterator.basic]

1

The `iterator` template may be used as a base class to ease the definition of required types for new iterators.

```
namespace std {
  template<class Category, class T, class Distance = ptrdiff_t,
    class Pointer = T*, class Reference = T&>
  struct iterator {
    typedef T         value_type;
    typedef Distance  difference_type;
    typedef Pointer   pointer;
    typedef Reference reference;
    typedef Category  iterator_category;
  };
}
```

24.183.1 Standard iterator tags [std.iterator.tags]

1

It is often desirable for a function template specialization to find out what is the most specific category of its iterator argument, so that the function can select the most efficient algorithm at compile time. To facilitate this, the library introduces *category tag* classes which are used as compile time tags for algorithm selection. They are: `input_iterator_tag`, `output_iterator_tag`, `forward_iterator_tag`, `bidirectional_iterator_tag` and `random_access_iterator_tag`. For every iterator of type `Iterator`, `iterator_traits<Iterator>::iterator_category` shall be defined to be the most specific category tag that describes the iterator's behavior.

```
namespace std {
  struct input_iterator_tag { };
  struct output_iterator_tag { };
  struct forward_iterator_tag: public input_iterator_tag { };
  struct bidirectional_iterator_tag: public forward_iterator_tag { };
  struct random_access_iterator_tag: public bidirectional_iterator_tag { };
}
```

1 [*Example:* For a program-defined iterator `BinaryTreeIterator`, it could be included into the bidirectional iterator category by specializing the `iterator_traits` template:

```
template<class T> struct iterator_traits<BinaryTreeIterator<T> > {
  typedef std::ptrdiff_t difference_type;
  typedef T value_type;
  typedef T* pointer;
  typedef T& reference;
  typedef bidirectional_iterator_tag iterator_category;
};
```

Typically, however, it would be easier to derive `BinaryTreeIterator<T>` from `iterator<bidirectional_iterator_tag,T,ptrdiff_t,T*,T&>`. — *end example*]

2 [*Example:* If `evolve()` is well defined for bidirectional iterators, but can be implemented more efficiently for random access iterators, then the implementation is as follows:

```
template <class BidirectionalIterator>
inline void
evolve(BidirectionalIterator first, BidirectionalIterator last) {
  evolve(first, last,
    typename iterator_traits<BidirectionalIterator>::iterator_category());
}

template <class BidirectionalIterator>
void evolve(BidirectionalIterator first, BidirectionalIterator last,
  bidirectional_iterator_tag) {
  // more generic, but less efficient algorithm
}

template <class RandomAccessIterator>
void evolve(RandomAccessIterator first, RandomAccessIterator last,
  random_access_iterator_tag) {
  // more efficient, but less generic algorithm
}
```

— *end example*]

3 [*Example:* If a C++ program wants to define a bidirectional iterator for some data structure containing `double` and such that it works on a large memory model of the implementation, it can do so with:

```
class MyIterator :
  public iterator<bidirectional_iterator_tag, double, long, T*, T&> {
  // code implementing ++, etc.
};
```

4 Then there is no need to specialize the `iterator_traits` template. — *end example*]

24.185.1 Iterator operations [iterator.operations]

1

Since only random access iterators provide `+` and `-` operators, the library provides two function templates `advance` and `distance`. These function templates use `+` and `-` for random access iterators (and are, therefore, constant time for them); for input, forward and bidirectional iterators they use `++` to provide linear time implementations.

```
template <class InputIterator, class Distance>
  void advance(InputIterator& i, Distance n);
```

1

Requires: `n` shall be negative only for bidirectional and random access iterators.

1

Effects: Increments (or decrements for negative `n`) iterator reference `i` by `n`.

```
template<class InputIterator>
    typename iterator_traits<InputIterator>::difference_type
       distance(InputIterator first, InputIterator last);
```

1 *Effects:* If `InputIterator` meets the requirements of random access iterator, returns `(last - first)`; otherwise, returns the number of increments needed to get from `first` to `last`.

2

Requires: If `InputIterator` meets the requirements of random access iterator, `last` shall be reachable from `first` or `first` shall be reachable from `last`; otherwise, `last` shall be reachable from `first`.

```
template <class ForwardIterator>
  ForwardIterator next(ForwardIterator x,
    typename std::iterator_traits<ForwardIterator>::difference_type n = 1);
```

1 *Effects:* Equivalent to `advance(x, n); return x;`

```
template <class BidirectionalIterator>
  BidirectionalIterator prev(BidirectionalIterator x,
    typename std::iterator_traits<BidirectionalIterator>::difference_type n = 1);
```

2 *Effects:* Equivalent to `advance(x, -n); return x;`

24.197 Iterator adaptors [predef.iterators]

24.197.1 Reverse iterators [reverse.iterators]

1

Class template `reverse_iterator` is an iterator adaptor that iterates from the end of the sequence defined by its underlying iterator to the beginning of that sequence. The fundamental relation between a reverse iterator and its corresponding iterator `i` is established by the identity: `&*(reverse_iterator(i)) == &*(i - 1)`.

24.203.0.1 Class template `reverse_iterator` [reverse.iterator]

```
namespace std {
  template <class Iterator>
  class reverse_iterator : public
        iterator<typename iterator_traits<Iterator>::iterator_category,
        typename iterator_traits<Iterator>::value_type,
        typename iterator_traits<Iterator>::difference_type,
        typename iterator_traits<Iterator>::pointer,
        typename iterator_traits<Iterator>::reference> {
  public:
    typedef Iterator                                          iterator_type;
    typedef typename iterator_traits<Iterator>::difference_type difference_type;
    typedef typename iterator_traits<Iterator>::reference       reference;
    typedef typename iterator_traits<Iterator>::pointer         pointer;

    reverse_iterator();
    explicit reverse_iterator(Iterator x);
    template <class U> reverse_iterator(const reverse_iterator<U>& u);
    template <class U> reverse_iterator& operator=(const reverse_iterator<U>& u);

    Iterator base() const;      // explicit
    reference operator*() const;
    pointer   operator->() const;

    reverse_iterator& operator++();
    reverse_iterator  operator++(int);
    reverse_iterator& operator--();
    reverse_iterator  operator--(int);

    reverse_iterator  operator+ (difference_type n) const;
```

```
    reverse_iterator& operator+=(difference_type n);
    reverse_iterator  operator- (difference_type n) const;
    reverse_iterator& operator-=(difference_type n);
    unspecified  operator[](difference_type n) const;
  protected:
    Iterator current;
  };

  template <class Iterator1, class Iterator2>
    bool operator==(
      const reverse_iterator<Iterator1>& x,
      const reverse_iterator<Iterator2>& y);
  template <class Iterator1, class Iterator2>
    bool operator<(
      const reverse_iterator<Iterator1>& x,
      const reverse_iterator<Iterator2>& y);
  template <class Iterator1, class Iterator2>
    bool operator!=(
      const reverse_iterator<Iterator1>& x,
      const reverse_iterator<Iterator2>& y);
  template <class Iterator1, class Iterator2>
    bool operator>(
      const reverse_iterator<Iterator1>& x,
      const reverse_iterator<Iterator2>& y);
  template <class Iterator1, class Iterator2>
    bool operator>=(
      const reverse_iterator<Iterator1>& x,
      const reverse_iterator<Iterator2>& y);
  template <class Iterator1, class Iterator2>
    bool operator<=(
      const reverse_iterator<Iterator1>& x,
      const reverse_iterator<Iterator2>& y);
  template <class Iterator1, class Iterator2>
    auto operator-(
      const reverse_iterator<Iterator1>& x,
      const reverse_iterator<Iterator2>& y) -> decltype(y.base() - x.base());
  template <class Iterator>
    reverse_iterator<Iterator> operator+(
      typename reverse_iterator<Iterator>::difference_type n,
      const reverse_iterator<Iterator>& x);

  template <class Iterator>
    reverse_iterator<Iterator> make_reverse_iterator(Iterator i);
}
```

24.203.0.2 `reverse_iterator` requirements [reverse.iter.requirements]

1 The template parameter `Iterator` shall meet all the requirements of a Bidirectional Iterator (24.165.1).

2 Additionally, `Iterator` shall meet the requirements of a Random Access Iterator (24.167.1) if any of the members `operator+` (24.203.0.3.8), `operator-` (24.203.0.3.10), `operator+=` (24.203.0.3.9), `operator-=` (24.203.0.3.11), `operator []` (24.203.0.3.12), or the global operators `operator<` (24.203.0.3.14), `operator>` (24.203.0.3.16), `operator <=` (24.203.0.3.18), `operator>=` (24.203.0.3.17), `operator-` (24.203.0.3.19) or `operator+` (24.203.0.3.20) are referenced in a way that requires instantiation (14.1644.1).

24.203.0.3 `reverse_iterator` operations [reverse.iter.ops]

24.203.0.3.1 `reverse_iterator` constructor [reverse.iter.cons]

```
reverse_iterator();
```

1 *Effects:* Value initializes `current`. Iterator operations applied to the resulting iterator have defined behavior if and only if the corresponding operations are defined on a value-initialized iterator of type `Iterator`.

```
explicit reverse_iterator(Iterator x);
```

2 *Effects:* Initializes `current` with `x`.

```
template <class U> reverse_iterator(const reverse_iterator<U> &u);
```

3 *Effects:* Initializes `current` with `u.current`.

24.203.0.3.2 `reverse_iterator::operator=` [reverse.iter.op=]

```
template <class U>
reverse_iterator&
  operator=(const reverse_iterator<U>& u);
```

1 *Effects:* Assigns `u.base()` to current.

2 *Returns:* `*this`.

24.203.0.3.3 Conversion [reverse.iter.conv]

```
Iterator base() const;          // explicit
```

1 *Returns:* `current`.

24.203.0.3.4 `operator*` [reverse.iter.op.star]

```
reference operator*() const;
```

1 *Effects:*

```
Iterator tmp = current;
return *--tmp;
```

24.203.0.3.5 `operator->` [reverse.iter.opref]

```
pointer operator->() const;
```

1 *Returns:* `std::addressof(operator*())`.

24.203.0.3.6 `operator++` [reverse.iter.op++]

```
reverse_iterator& operator++();
```

1 *Effects:* `--current;`

2 *Returns:* `*this`.

```
reverse_iterator operator++(int);
```

3 *Effects:*

```
reverse_iterator tmp = *this;
--current;
return tmp;
```

24.203.0.3.7 `operator--` [reverse.iter.op--]

```
reverse_iterator& operator--();
```

1 *Effects:* `++current`

2 *Returns:* `*this`.

```
reverse_iterator operator--(int);
```

3 *Effects:*

```
reverse_iterator tmp = *this;
++current;
return tmp;
```

24.203.0.3.8 `operator+` [reverse.iter.op+]

```
reverse_iterator
operator+(typename reverse_iterator<Iterator>::difference_type n) const;
```

1 *Returns:* `reverse_iterator(current-n)`.

24.203.0.3.9 `operator+=` [reverse.iter.op+=]

```
reverse_iterator&
operator+=(typename reverse_iterator<Iterator>::difference_type n);
```

1 *Effects:* `current -= n;`

2 *Returns:* `*this`.

24.203.0.3.10 `operator-` [reverse.iter.op-]

```
reverse_iterator
operator-(typename reverse_iterator<Iterator>::difference_type n) const;
```

1 *Returns:* `reverse_iterator(current+n)`.

24.203.0.3.11 `operator-=` [reverse.iter.op-=]

```
reverse_iterator&
operator-=(typename reverse_iterator<Iterator>::difference_type n);
```

1 *Effects:* `current += n;`

2 *Returns:* `*this`.

24.203.0.3.12 `operator[]` [reverse.iter.opindex]

```
unspecified  operator[](
    typename reverse_iterator<Iterator>::difference_type n) const;
```

1 *Returns:* `current[-n-1]`.

24.203.0.3.13 `operator==` [reverse.iter.op==]

```
template <class Iterator1, class Iterator2>
  bool operator==(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
```

1 *Returns:* `x.current == y.current`.

24.203.0.3.14 `operator<` [reverse.iter.op<]

```
template <class Iterator1, class Iterator2>
  bool operator<(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
```

1 *Returns:* `x.current > y.current`.

24.203.0.3.15 `operator!=` [reverse.iter.op!=]

```
template <class Iterator1, class Iterator2>
  bool operator!=(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
```

1 *Returns:* `x.current != y.current`.

24.203.0.3.16 `operator>` [reverse.iter.op>]

```
template <class Iterator1, class Iterator2>
  bool operator>(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
```

1 *Returns:* `x.current < y.current`.

24.203.0.3.17 `operator>=` [reverse.iter.op>=]

```
template <class Iterator1, class Iterator2>
  bool operator>=(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
```

1 *Returns:* `x.current <= y.current`.

24.203.0.3.18 `operator<=` [reverse.iter.op<=]

```
template <class Iterator1, class Iterator2>
  bool operator<=(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
```

1 *Returns:* `x.current >= y.current`.

24.203.0.3.19 `operator-` [reverse.iter.opdiff]

```
template <class Iterator1, class Iterator2>
    auto operator-(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y) -> decltype(y.base() - x.base());
```

1 *Returns:* `y.current - x.current`.

24.203.0.3.20 `operator+` [reverse.iter.opsum]

```
template <class Iterator>
  reverse_iterator<Iterator> operator+(
    typename reverse_iterator<Iterator>::difference_type n,
    const reverse_iterator<Iterator>& x);
```

1 *Returns:* `reverse_iterator<Iterator> (x.current - n)`.

24.203.0.3.21 Non-member function `make_reverse_iterator()` [reverse.iter.make]

```
template <class Iterator>
  reverse_iterator<Iterator> make_reverse_iterator(Iterator i);
```

1 *Returns:* `reverse_iterator<Iterator>(i)`.

24.203.1 Insert iterators [insert.iterators]

1 To make it possible to deal with insertion in the same way as writing into an array, a special kind of iterator adaptors, called *insert iterators*, are provided in the library. With regular iterator classes,

```
while (first != last) *result++ = *first++;
```

causes a range `[first,last)` to be copied into a range starting with result. The same code with `result` being an insert iterator will insert corresponding elements into the container. This device allows all of the copying algorithms in the library to work in the *insert mode* instead of the *regular overwrite* mode.

2

An insert iterator is constructed from a container and possibly one of its iterators pointing to where insertion takes place if it is neither at the beginning nor at the end of the container. Insert iterators satisfy the requirements of output iterators. `operator*` returns the insert iterator itself. The assignment `operator=(const T& x)` is defined on insert iterators to allow writing into them, it inserts `x` right before where the insert iterator is pointing. In other words, an insert iterator is like a cursor pointing into the container where the insertion takes place. `back_insert_iterator` inserts elements at the end of a container, `front_insert_iterator` inserts elements at the beginning of a container, and `insert_iterator` inserts elements where the iterator points to in a container. `back_inserter`, `front_inserter`, and `inserter` are three functions making the insert iterators out of a container.

24.205.0.1 Class template `back_insert_iterator` [back.insert.iterator]

```
namespace std {
  template <class Container>
  class back_insert_iterator :
    public iterator<output_iterator_tag,void,void,void,void> {
  protected:
    Container* container;

  public:
    typedef Container container_type;
    explicit back_insert_iterator(Container& x);
    back_insert_iterator<Container>&
      operator=(const typename Container::value_type& value);
    back_insert_iterator<Container>&
      operator=(typename Container::value_type&& value);

    back_insert_iterator<Container>& operator*();
    back_insert_iterator<Container>& operator++();
    back_insert_iterator<Container>  operator++(int);
  };

  template <class Container>
    back_insert_iterator<Container> back_inserter(Container& x);
}
```

24.205.0.2 `back_insert_iterator` operations [back.insert.iter.ops]

24.205.0.2.1 `back_insert_iterator` constructor [back.insert.iter.cons]

```
explicit back_insert_iterator(Container& x);
```

1 *Effects:* Initializes `container` with `std::addressof(x)`.

24.205.0.2.2 `back_insert_iterator::operator=` [back.insert.iter.op=]

```
back_insert_iterator<Container>&
  operator=(const typename Container::value_type& value);
```

1 *Effects:* `container->push_back(value);`

2 *Returns:* `*this`.

```
back_insert_iterator<Container>&
  operator=(typename Container::value_type&& value);
```

3 *Effects:* `container->push_back(std::move(value));`

4 *Returns:* `*this`.

24.205.0.2.3 `back_insert_iterator::operator*` [back.insert.iter.op*]

```
back_insert_iterator<Container>& operator*();
```

1 *Returns:* `*this`.

24.205.0.2.4 `back_insert_iterator::operator++` [back.insert.iter.op++]

```
back_insert_iterator<Container>& operator++();
back_insert_iterator<Container>  operator++(int);
```

1 *Returns:* `*this`.

24.205.0.2.5 `back_inserter` [back.inserter]

```
template <class Container>
  back_insert_iterator<Container> back_inserter(Container& x);
```

1 *Returns:* `back_insert_iterator<Container>(x)`.

24.205.0.3 Class template `front_insert_iterator` [front.insert.iterator]

```
namespace std {
  template <class Container>
  class front_insert_iterator :
    public iterator<output_iterator_tag,void,void,void,void> {
  protected:
    Container* container;

  public:
    typedef Container container_type;
    explicit front_insert_iterator(Container& x);
    front_insert_iterator<Container>&
      operator=(const typename Container::value_type& value);
    front_insert_iterator<Container>&
      operator=(typename Container::value_type&& value);

    front_insert_iterator<Container>& operator*();
    front_insert_iterator<Container>& operator++();
    front_insert_iterator<Container>  operator++(int);
  };
```

```
  template <class Container>
    front_insert_iterator<Container> front_inserter(Container& x);
}
```

24.205.0.4 `front_insert_iterator` operations [front.insert.iter.ops]

24.205.0.4.1 `front_insert_iterator` constructor [front.insert.iter.cons]

```
explicit front_insert_iterator(Container& x);
```

1 *Effects:* Initializes `container` with `std::addressof(x)`.

24.205.0.4.2 `front_insert_iterator::operator=` [front.insert.iter.op=]

```
front_insert_iterator<Container>&
  operator=(const typename Container::value_type& value);
```

1 *Effects:* `container->push_front(value);`

2 *Returns:* `*this`.

```
front_insert_iterator<Container>&
  operator=(typename Container::value_type&& value);
```

3 *Effects:* `container->push_front(std::move(value));`

4 *Returns:* `*this`.

24.205.0.4.3 `front_insert_iterator::operator*` [front.insert.iter.op*]

```
front_insert_iterator<Container>& operator*();
```

1 *Returns:* `*this`.

24.205.0.4.4 `front_insert_iterator::operator++` [front.insert.iter.op++]

```
front_insert_iterator<Container>& operator++();
front_insert_iterator<Container>  operator++(int);
```

1 *Returns:* `*this`.

24.205.0.4.5 `front_inserter` [front.inserter]

```
template <class Container>
  front_insert_iterator<Container> front_inserter(Container& x);
```

1 *Returns:* `front_insert_iterator<Container>(x)`.

24.205.0.5 Class template `insert_iterator` [insert.iterator]

```
namespace std {
  template <class Container>
  class insert_iterator :
    public iterator<output_iterator_tag,void,void,void,void> {
  protected:
    Container* container;
    typename Container::iterator iter;

  public:
    typedef Container container_type;
    insert_iterator(Container& x, typename Container::iterator i);
    insert_iterator<Container>&
      operator=(const typename Container::value_type& value);
```

```
    insert_iterator<Container>&
      operator=(typename Container::value_type&& value);

    insert_iterator<Container>& operator*();
    insert_iterator<Container>& operator++();
    insert_iterator<Container>& operator++(int);
  };

  template <class Container>
    insert_iterator<Container> inserter(Container& x, typename Container::iterator i);
}
```

24.205.0.6 insert_iterator operations [insert.iter.ops]

24.205.0.6.1 insert_iterator constructor [insert.iter.cons]

```
insert_iterator(Container& x, typename Container::iterator i);
```

1 *Effects:* Initializes `container` with `std::addressof(x)` and `iter` with `i`.

24.205.0.6.2 insert_iterator::operator= [insert.iter.op=]

```
insert_iterator<Container>&
  operator=(const typename Container::value_type& value);
```

1 *Effects:*

```
iter = container->insert(iter, value);
++iter;
```

2 *Returns:* `*this`.

```
insert_iterator<Container>&
  operator=(typename Container::value_type&& value);
```

3 *Effects:*

```
iter = container->insert(iter, std::move(value));
++iter;
```

4 *Returns:* `*this`.

24.205.0.6.3 insert_iterator::operator* [insert.iter.op*]

```
insert_iterator<Container>& operator*();
```

1 *Returns:* `*this`.

24.205.0.6.4 insert_iterator::operator++ [insert.iter.op++]

```
insert_iterator<Container>& operator++();
insert_iterator<Container>& operator++(int);
```

1 *Returns:* `*this`.

24.205.0.6.5 inserter [inserter]

```
template <class Container>
  insert_iterator<Container> inserter(Container& x, typename Container::iterator i);
```

1 *Returns:* `insert_iterator<Container>(x, i)`.

24.205.1 Move iterators [move.iterators]

1

Class template `move_iterator` is an iterator adaptor with the same behavior as the underlying iterator except that its indirection operator implicitly converts the value returned by the underlying iterator's indirection operator to an rvalue reference. Some generic algorithms can be called with move iterators to replace copying with moving.

1 [*Example:*

```
list<string> s;
// populate the list s
vector<string> v1(s.begin(), s.end());          // copies strings into v1
vector<string> v2(make_move_iterator(s.begin()),
                  make_move_iterator(s.end())); // moves strings into v2
```

— *end example*]

24.206.0.1 Class template `move_iterator` [move.iterator]

```
namespace std {
  template <class Iterator>
  class move_iterator {
  public:
    typedef Iterator                                              iterator_type;
    typedef typename iterator_traits<Iterator>::difference_type   difference_type;
    typedef Iterator                                              pointer;
    typedef typename iterator_traits<Iterator>::value_type        value_type;
    typedef typename iterator_traits<Iterator>::iterator_category iterator_category;
    typedef value_type&&                                          reference;

    move_iterator();
    explicit move_iterator(Iterator i);
    template <class U> move_iterator(const move_iterator<U>& u);
    template <class U> move_iterator& operator=(const move_iterator<U>& u);

    iterator_type base() const;
    reference operator*() const;
    pointer operator->() const;

    move_iterator& operator++();
    move_iterator operator++(int);
    move_iterator& operator--();
    move_iterator operator--(int);

    move_iterator operator+(difference_type n) const;
    move_iterator& operator+=(difference_type n);
    move_iterator operator-(difference_type n) const;
    move_iterator& operator-=(difference_type n);
    unspecified  operator[](difference_type n) const;

  private:
    Iterator current;    // exposition only
  };

  template <class Iterator1, class Iterator2>
    bool operator==(
```

```
      const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
  template <class Iterator1, class Iterator2>
    bool operator!=(
      const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
  template <class Iterator1, class Iterator2>
    bool operator<(
      const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
  template <class Iterator1, class Iterator2>
    bool operator<=(
      const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
  template <class Iterator1, class Iterator2>
    bool operator>(
      const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
  template <class Iterator1, class Iterator2>
    bool operator>=(
      const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);

  template <class Iterator1, class Iterator2>
    auto operator-(
      const move_iterator<Iterator1>& x,
      const move_iterator<Iterator2>& y) -> decltype(x.base() - y.base());
  template <class Iterator>
    move_iterator<Iterator> operator+(
      typename move_iterator<Iterator>::difference_type n, const move_iterator<Iterator>& x);
  template <class Iterator>
    move_iterator<Iterator> make_move_iterator(Iterator i);
}
```

24.206.0.2 `move_iterator` requirements [move.iter.requirements]

1 The template parameter `Iterator` shall meet the requirements for an Input Iterator (24.100.1). Additionally, if any of the bidirectional or random access traversal functions are instantiated, the template parameter shall meet the requirements for a Bidirectional Iterator (24.165.1) or a Random Access Iterator (24.167.1), respectively.

24.206.0.3 `move_iterator` operations [move.iter.ops]

24.206.0.3.1 `move_iterator` constructors [move.iter.op.const]

```
move_iterator();
```

1 *Effects:* Constructs a `move_iterator`, value initializing `current`. Iterator operations applied to the resulting iterator have defined behavior if and only if the corresponding operations are defined on a value-initialized iterator of type `Iterator`.

```
explicit move_iterator(Iterator i);
```

2 *Effects:* Constructs a `move_iterator`, initializing `current` with `i`.

```
template <class U> move_iterator(const move_iterator<U>& u);
```

3 *Effects:* Constructs a `move_iterator`, initializing `current` with `u.base()`.

4 *Requires:* `U` shall be convertible to `Iterator`.

24.206.0.3.2 `move_iterator::operator=` [move.iter.op=]

```
template <class U> move_iterator& operator=(const move_iterator<U>& u);
```

1 *Effects:* Assigns `u.base()` to `current`.

2 *Requires:* `U` shall be convertible to `Iterator`.

24.206.0.3.3 `move_iterator` conversion [move.iter.op.conv]

```
Iterator base() const;
```

1 *Returns:* `current`.

24.206.0.3.4 `move_iterator::operator*` [move.iter.op.star]

```
reference operator*() const;
```

1 *Returns:* `std::move(`*current).

24.206.0.3.5 `move_iterator::operator->` [move.iter.op.ref]

```
pointer operator->() const;
```

1 *Returns:* `current`.

24.206.0.3.6 `move_iterator::operator++` [move.iter.op.incr]

```
move_iterator& operator++();
```

1 *Effects:* `++current`.

2 *Returns:* `*this`.

```
move_iterator operator++(int);
```

3 *Effects:*

```
move_iterator tmp = *this;
++current;
return tmp;
```

24.206.0.3.7 `move_iterator::operator--` [move.iter.op.decr]

```
move_iterator& operator--();
```

1 *Effects:* `--current`.

2 *Returns:* `*this`.

```
move_iterator operator--(int);
```

3 *Effects:*

```
move_iterator tmp = *this;
--current;
return tmp;
```

24.206.0.3.8 `move_iterator::operator+` [move.iter.op.+]

```
move_iterator operator+(difference_type n) const;
```

1 *Returns:* `move_iterator(current + n)`.

24.206.0.3.9 `move_iterator::operator+=` [move.iter.op.+=]

```
move_iterator& operator+=(difference_type n);
```

1 *Effects:* `current += n`.

2 *Returns:* `*this`.

24.206.0.3.10 `move_iterator::operator-` [move.iter.op.-]

```
move_iterator operator-(difference_type n) const;
```

1 *Returns:* `move_iterator(current - n)`.

24.206.0.3.11 `move_iterator::operator-=` [move.iter.op.-=]

```
move_iterator& operator-=(difference_type n);
```

1 *Effects:* `current -= n`.

2 *Returns:* `*this`.

24.206.0.3.12 `move_iterator::operator[]` [move.iter.op.index]

```
unspecified  operator[](difference_type n) const;
```

1 *Returns:* `std::move(current[n])`.

24.206.0.3.13 `move_iterator` comparisons [move.iter.op.comp]

```
template <class Iterator1, class Iterator2>
bool operator==(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
```

1 *Returns:* `x.base() == y.base()`.

```
template <class Iterator1, class Iterator2>
bool operator!=(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
```

2 *Returns:* `!(x == y)`.

```
template <class Iterator1, class Iterator2>
bool operator<(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
```

3 *Returns:* `x.base() < y.base()`.

```
template <class Iterator1, class Iterator2>
bool operator<=(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
```

4 *Returns:* `!(y < x)`.

```
template <class Iterator1, class Iterator2>
bool operator>(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
```

5 *Returns:* `y < x`.

```
template <class Iterator1, class Iterator2>
bool operator>=(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
```

6 *Returns:* `!(x < y)`.

24.206.0.3.14 `move_iterator` non-member functions [move.iter.nonmember]

```
template <class Iterator1, class Iterator2>
    auto operator-(
    const move_iterator<Iterator1>& x,
    const move_iterator<Iterator2>& y) -> decltype(x.base() - y.base());
```

1 *Returns:* `x.base() - y.base()`.

```
template <class Iterator>
  move_iterator<Iterator> operator+(
    typename move_iterator<Iterator>::difference_type n, const move_iterator<Iterator>& x);
```

2 *Returns:* `x + n`.

```
template <class Iterator>
move_iterator<Iterator> make_move_iterator(Iterator i);
```

3 *Returns:* `move_iterator<Iterator>(i)`.

24.207 Stream iterators [stream.iterators]

1 To make it possible for algorithmic templates to work directly with input/output streams, appropriate iterator-like class templates are provided.

[*Example:*

```
partial_sum(istream_iterator<double, char>(cin),
  istream_iterator<double, char>(),
  ostream_iterator<double, char>(cout, "\n"));
```

reads a file containing floating point numbers from `cin`, and prints the partial sums onto `cout`. — *end example*]

24.207.1 Class template `istream_iterator` [istream.iterator]

1

The class template `istream_iterator` is an input iterator (24.100.1) that reads (using `operator>>`) successive elements from the input stream for which it was constructed. After it is constructed, and every time `++` is used, the iterator reads and stores a value of `T`. If the iterator fails to read and store a value of `T` (`fail()` on the stream returns `true`), the iterator becomes equal to the *end-of-stream* iterator value. The constructor with no arguments `istream_iterator()` always constructs an end-of-stream input iterator object, which is the only legitimate iterator to be used for the end condition. The result of `operator*` on an end-of-stream iterator is not defined. For any other iterator value a `const T&` is returned. The result of `operator->` on an end-of-stream iterator is not defined. For any other iterator value a `const T*` is returned. The behavior of a program that applies `operator++()` to an end-of-stream iterator is undefined. It is impossible to store things into istream iterators.

1

Two end-of-stream iterators are always equal. An end-of-stream iterator is not equal to a non-end-of-stream iterator. Two non-end-of-stream iterators are equal when they are constructed from the same stream.

```
namespace std {
  template <class T, class charT = char, class traits = char_traits<charT>,
      class Distance = ptrdiff_t>
  class istream_iterator:
    public iterator<input_iterator_tag, T, Distance, const T*, const T&> {
  public:
    typedef charT char_type;
    typedef traits traits_type;
    typedef basic_istream<charT,traits> istream_type;
    see below istream_iterator();
    istream_iterator(istream_type& s);
    istream_iterator(const istream_iterator& x) = default;
   ~istream_iterator() = default;

    const T& operator*() const;
    const T* operator->() const;
    istream_iterator<T,charT,traits,Distance>& operator++();
    istream_iterator<T,charT,traits,Distance>  operator++(int);
```

```
  private:
    basic_istream<charT,traits>* in_stream; // exposition only
    T value;                                // exposition only
  };

  template <class T, class charT, class traits, class Distance>
    bool operator==(const istream_iterator<T,charT,traits,Distance>& x,
            const istream_iterator<T,charT,traits,Distance>& y);
  template <class T, class charT, class traits, class Distance>
    bool operator!=(const istream_iterator<T,charT,traits,Distance>& x,
            const istream_iterator<T,charT,traits,Distance>& y);
}
```

24.215.0.1 `istream_iterator` constructors and destructor [istream.iterator.cons]

see below `istream_iterator();`

1 *Effects:* Constructs the end-of-stream iterator. If `T` is a literal type, then this constructor shall be a `constexpr` constructor.

2 *Postcondition:* `in_stream == 0`.

`istream_iterator(istream_type& s);`

3 *Effects:* Initializes *in_stream* with `&s`. *value* may be initialized during construction or the first time it is referenced.

1 *Postcondition:* `in_stream == &s`.

`istream_iterator(const istream_iterator& x) = default;`

2 *Effects:* Constructs a copy of `x`. If `T` is a literal type, then this constructor shall be a trivial copy constructor.

3 *Postcondition:* `in_stream == x.in_stream`.

`~istream_iterator() = default;`

4 *Effects:* The iterator is destroyed. If `T` is a literal type, then this destructor shall be a trivial destructor.

24.218.0.1 `istream_iterator` operations [istream.iterator.ops]

`const T& operator*() const;`

1 *Returns: value.*

`const T* operator->() const;`

2 *Returns:* `&(operator*())`.

`istream_iterator<T,charT,traits,Distance>& operator++();`

3 *Requires:* `in_stream != 0`.

4 *Effects:* `*in_stream >>value`.

5 *Returns:* `*this`.

`istream_iterator<T,charT,traits,Distance> operator++(int);`

6 *Requires:* `in_stream != 0`.

7 *Effects:*

```
istream_iterator<T,charT,traits,Distance> tmp = *this;
*in_stream >> value;
return (tmp);
```

```
template <class T, class charT, class traits, class Distance>
  bool operator==(const istream_iterator<T,charT,traits,Distance> &x,
                  const istream_iterator<T,charT,traits,Distance> &y);
```

8 *Returns:* `x.in_stream == y.in_stream`.

```
template <class T, class charT, class traits, class Distance>
  bool operator!=(const istream_iterator<T,charT,traits,Distance> &x,
                  const istream_iterator<T,charT,traits,Distance> &y);
```

9 *Returns:* `!(x == y)`

24.218.1 Class template `ostream_iterator` [ostream.iterator]

1

`ostream_iterator` writes (using `operator<<`) successive elements onto the output stream from which it was constructed. If it was constructed with `charT*` as a constructor argument, this string, called a *delimiter string*, is written to the stream after every `T` is written. It is not possible to get a value out of the output iterator. Its only use is as an output iterator in situations like

```
while (first != last)
  *result++ = *first++;
```

1 `ostream_iterator` is defined as:

```
namespace std {
  template <class T, class charT = char, class traits = char_traits<charT> >
  class ostream_iterator:
    public iterator<output_iterator_tag, void, void, void, void> {
  public:
    typedef charT char_type;
    typedef traits traits_type;
    typedef basic_ostream<charT,traits> ostream_type;
    ostream_iterator(ostream_type& s);
    ostream_iterator(ostream_type& s, const charT* delimiter);
    ostream_iterator(const ostream_iterator<T,charT,traits>& x);
   ~ostream_iterator();
    ostream_iterator<T,charT,traits>& operator=(const T& value);

    ostream_iterator<T,charT,traits>& operator*();
    ostream_iterator<T,charT,traits>& operator++();
    ostream_iterator<T,charT,traits>& operator++(int);
  private:
    basic_ostream<charT,traits>* out_stream;  // exposition only
    const charT* delim;                       // exposition only
  };
}
```

24.219.0.1 `ostream_iterator` constructors and destructor [ostream.iterator.cons.des]

`ostream_iterator(ostream_type& s);`

1 *Effects:* Initializes *out_stream* with `&s` and *delim* with null.

`ostream_iterator(ostream_type& s, const charT* delimiter);`

2 *Effects:* Initializes *out_stream* with `&s` and *delim* with `delimiter`.

`ostream_iterator(const ostream_iterator& x);`

3 *Effects:* Constructs a copy of `x`.

`~ostream_iterator();`

4 *Effects:* The iterator is destroyed.

24.219.0.2 `ostream_iterator` operations [ostream.iterator.ops]

`ostream_iterator& operator=(const T& value);`

1 *Effects:*

```
*out_stream << value;
if (delim != 0)
  *out_stream << delim;
return *this;
```

`ostream_iterator& operator*();`

2 *Returns:* `*this`.

```
ostream_iterator& operator++();
ostream_iterator& operator++(int);
```

3 *Returns:* `*this`.

24.219.1 Class template `istreambuf_iterator` [istreambuf.iterator]

1 The class template `istreambuf_iterator` defines an input iterator (24.100.1) that reads successive *characters* from the streambuf for which it was constructed. `operator*` provides access to the current input character, if any. [*Note:* `operator->` may return a proxy. — *end note*] Each time `operator++` is evaluated, the iterator advances to the next input character. If the end of stream is reached (`streambuf_type::sgetc()` returns `traits::eof()`), the iterator becomes equal to the *end-of-stream* iterator value. The default constructor `istreambuf_iterator()` and the constructor `istreambuf_iterator(0)` both construct an end-of-stream iterator object suitable for use as an end-of-range. All specializations of `istreambuf_iterator` shall have a trivial copy constructor, a `constexpr` default constructor, and a trivial destructor.

2 The result of `operator*()` on an end-of-stream iterator is undefined. For any other iterator value a `char_type` value is returned. It is impossible to assign a character via an input iterator.

```
namespace std {
  template<class charT, class traits = char_traits<charT> >
  class istreambuf_iterator
     : public iterator<input_iterator_tag, charT,
                       typename traits::off_type, unspecified , charT> {
  public:
    typedef charT                          char_type;
    typedef traits                         traits_type;
```

```
    typedef typename traits::int_type     int_type;
    typedef basic_streambuf<charT,traits> streambuf_type;
    typedef basic_istream<charT,traits>   istream_type;

    class proxy;                          // exposition only

    constexpr istreambuf_iterator() noexcept;
    istreambuf_iterator(const istreambuf_iterator&) noexcept = default;
    ~istreambuf_iterator() = default;
    istreambuf_iterator(istream_type& s) noexcept;
    istreambuf_iterator(streambuf_type* s) noexcept;
    istreambuf_iterator(const proxy& p) noexcept;
    charT operator*() const;
    pointer operator->() const;
    istreambuf_iterator<charT,traits>& operator++();
    proxy operator++(int);
    bool equal(const istreambuf_iterator& b) const;
  private:
    streambuf_type* sbuf_;                // exposition only
  };

  template <class charT, class traits>
    bool operator==(const istreambuf_iterator<charT,traits>& a,
            const istreambuf_iterator<charT,traits>& b);
  template <class charT, class traits>
    bool operator!=(const istreambuf_iterator<charT,traits>& a,
            const istreambuf_iterator<charT,traits>& b);
}
```

24.219.1.1 Class template `istreambuf_iterator::proxy` [istreambuf.iterator::proxy]

```
namespace std {
  template <class charT, class traits = char_traits<charT> >
  class istreambuf_iterator<charT, traits>::proxy { // exposition only
    charT keep_;
    basic_streambuf<charT,traits>* sbuf_;
    proxy(charT c, basic_streambuf<charT,traits>* sbuf)
      : keep_(c), sbuf_(sbuf) { }
  public:
    charT operator*() { return keep_; }
  };
}
```

1 Class `istreambuf_iterator<charT,traits>::proxy` is for exposition only. An implementation is permitted to provide equivalent functionality without providing a class with this name. Class `istreambuf_iterator<charT, traits>::proxy` provides a temporary placeholder as the return value of the post-increment operator (`operator++`). It keeps the character pointed to by the previous value of the iterator for some possible future access to get the character.

24.219.1.2 `istreambuf_iterator` constructors [istreambuf.iterator.cons]

```
constexpr istreambuf_iterator() noexcept;
```

1 *Effects:* Constructs the end-of-stream iterator.

```
istreambuf_iterator(basic_istream<charT,traits>& s) noexcept;
istreambuf_iterator(basic_streambuf<charT,traits>* s) noexcept;
```

2 *Effects:* Constructs an `istreambuf_iterator<>` that uses the `basic_streambuf<>` object `*(s.rdbuf())`, or `*s`, respectively. Constructs an end-of-stream iterator if `s.rdbuf()` is null.

```
istreambuf_iterator(const proxy& p) noexcept;
```

3 *Effects:* Constructs a `istreambuf_iterator<>` that uses the `basic_streambuf<>` object pointed to by the `proxy` object's constructor argument `p`.

24.219.1.3 `istreambuf_iterator::operator*` [istreambuf.iterator::op*]

```
charT operator*() const
```

1 *Returns:* The character obtained via the `streambuf` member `sbuf_->sgetc()`.

24.219.1.4 `istreambuf_iterator::operator++` [istreambuf.iterator::op++]

```
istreambuf_iterator<charT,traits>&
    istreambuf_iterator<charT,traits>::operator++();
```

1 *Effects:* `sbuf_->sbumpc()`.

2 *Returns:* `*this`.

```
proxy istreambuf_iterator<charT,traits>::operator++(int);
```

3 *Returns:* `proxy(sbuf_->sbumpc(), sbuf_)`.

24.219.1.5 `istreambuf_iterator::equal` [istreambuf.iterator::equal]

```
bool equal(const istreambuf_iterator<charT,traits>& b) const;
```

1

Returns: `true` if and only if both iterators are at end-of-stream, or neither is at end-of-stream, regardless of what `streambuf` object they use.

24.220.0.1 `operator==` [istreambuf.iterator::op==]

```
template <class charT, class traits>
  bool operator==(const istreambuf_iterator<charT,traits>& a,
                  const istreambuf_iterator<charT,traits>& b);
```

1 *Returns:* `a.equal(b)`.

24.220.0.2 `operator!=` [istreambuf.iterator::op!=]

```
template <class charT, class traits>
  bool operator!=(const istreambuf_iterator<charT,traits>& a,
                  const istreambuf_iterator<charT,traits>& b);
```

1 *Returns:* `!a.equal(b)`.

24.220.1 Class template `ostreambuf_iterator` [ostreambuf.iterator]

```
namespace std {
  template <class charT, class traits = char_traits<charT> >
  class ostreambuf_iterator :
    public iterator<output_iterator_tag, void, void, void, void> {
  public:
    typedef charT                        char_type;
    typedef traits                       traits_type;
    typedef basic_streambuf<charT,traits> streambuf_type;
```

```
    typedef basic_ostream<charT,traits>   ostream_type;

  public:
    ostreambuf_iterator(ostream_type& s) noexcept;
    ostreambuf_iterator(streambuf_type* s) noexcept;
    ostreambuf_iterator& operator=(charT c);

    ostreambuf_iterator& operator*();
    ostreambuf_iterator& operator++();
    ostreambuf_iterator& operator++(int);
    bool failed() const noexcept;

  private:
    streambuf_type* sbuf_;                // exposition only
  };
}
```

1 The class template `ostreambuf_iterator` writes successive *characters* onto the output stream from which it was constructed. It is not possible to get a character value out of the output iterator.

24.220.1.1 `ostreambuf_iterator` constructors [ostreambuf.iter.cons]

`ostreambuf_iterator(ostream_type& s) noexcept;`

1 *Requires:* `s.rdbuf()` shall not be a null pointer.

2 *Effects:* Initializes `sbuf_` with `s.rdbuf()`.

`ostreambuf_iterator(streambuf_type* s) noexcept;`

3 *Requires:* `s` shall not be a null pointer.

4 *Effects:* Initializes `sbuf_` with `s`.

24.220.1.2 `ostreambuf_iterator` operations [ostreambuf.iter.ops]

```
ostreambuf_iterator<charT,traits>&
  operator=(charT c);
```

1 *Effects:* If `failed()` yields `false`, calls `sbuf_->sputc(c)`; otherwise has no effect.

2 *Returns:* `*this`.

`ostreambuf_iterator<charT,traits>& operator*();`

3 *Returns:* `*this`.

```
ostreambuf_iterator<charT,traits>& operator++();
ostreambuf_iterator<charT,traits>& operator++(int);
```

4 *Returns:* `*this`.

`bool failed() const noexcept;`

5 *Returns:* `true` if in any prior use of member `operator=`, the call to `sbuf_->sputc()` returned `traits::eof()`; or `false` otherwise.

24.221 range access [iterator.range]

1

In addition to being available via inclusion of the <iterator> header, the function templates in 24.221 are available when any of the following headers are included: <array>, <deque>, <forward_list>, <list>, <map>, <regex>, <set>, <string>, <unordered_map>, <unordered_set>, and <vector>.

```
template <class C> auto begin(C& c) -> decltype(c.begin());
template <class C> auto begin(const C& c) -> decltype(c.begin());
```

1 *Returns:* `c.begin()`.

```
template <class C> auto end(C& c) -> decltype(c.end());
template <class C> auto end(const C& c) -> decltype(c.end());
```

2 *Returns:* `c.end()`.

```
template <class T, size_t N> constexpr T* begin(T (&array)[N]) noexcept;
```

3 *Returns:* `array`.

```
template <class T, size_t N> constexpr T* end(T (&array)[N]) noexcept;
```

4 *Returns:* `array + N`.

```
template <class C> constexpr auto cbegin(const C& c) noexcept(noexcept(std::begin(c)))
  -> decltype(std::begin(c));
```

5 *Returns:* `std::begin(c)`.

```
template <class C> constexpr auto cend(const C& c) noexcept(noexcept(std::end(c)))
  -> decltype(std::end(c));
```

6 *Returns:* `std::end(c)`.

```
template <class C> auto rbegin(C& c) -> decltype(c.rbegin());
template <class C> auto rbegin(const C& c) -> decltype(c.rbegin());
```

7 *Returns:* `c.rbegin()`.

```
template <class C> auto rend(C& c) -> decltype(c.rend());
template <class C> auto rend(const C& c) -> decltype(c.rend());
```

8 *Returns:* `c.rend()`.

```
template <class T, size_t N> reverse_iterator<T*> rbegin(T (&array)[N]);
```

9 *Returns:* `reverse_iterator<T*>(array + N)`.

```
template <class T, size_t N> reverse_iterator<T*> rend(T (&array)[N]);
```

10 *Returns:* `reverse_iterator<T*>(array)`.

```
template <class E> reverse_iterator<const E*> rbegin(initializer_list<E> il);
```

11 *Returns:* `reverse_iterator<const E*>(il.end())`.

```
template <class E> reverse_iterator<const E*> rend(initializer_list<E> il);
```

12 *Returns:* `reverse_iterator<const E*>(il.begin())`.

```
template <class C> auto crbegin(const C& c) -> decltype(std::rbegin(c));
```

1 *Returns:* `std::rbegin(c)`.

```
template <class C> auto crend(const C& c) -> decltype(std::rend(c));
```

2 *Returns:* `std::rend(c)`.

25 Algorithms library [algorithms]

25.1 General [algorithms.general]

1 This Clause describes components that C++ programs may use to perform algorithmic operations on containers (Clause 23) and other sequences.

2 The following subclauses describe components for non-modifying sequence operation, modifying sequence operations, sorting and related operations, and algorithms from the ISO C library, as summarized in Table 112.

Table 112 — Algorithms library summary

	Subclause	Header(s)
25.23	Non-modifying sequence operations	
25.54	Mutating sequence operations	`<algorithm>`
25.98	Sorting and related operations	
25.205	C library algorithms	`<cstdlib>`

Header `<algorithm>` synopsis

```
#include <initializer_list>

namespace std {

  // 25.23, non-modifying sequence operations:
  template <class InputIterator, class Predicate>
    bool all_of(InputIterator first, InputIterator last, Predicate pred);
  template <class InputIterator, class Predicate>
    bool any_of(InputIterator first, InputIterator last, Predicate pred);
  template <class InputIterator, class Predicate>
    bool none_of(InputIterator first, InputIterator last, Predicate pred);

  template<class InputIterator, class Function>
    Function for_each(InputIterator first, InputIterator last, Function f);
  template<class InputIterator, class T>
    InputIterator find(InputIterator first, InputIterator last,
                       const T& value);
  template<class InputIterator, class Predicate>
    InputIterator find_if(InputIterator first, InputIterator last,
                          Predicate pred);
  template<class InputIterator, class Predicate>
    InputIterator find_if_not(InputIterator first, InputIterator last,
                              Predicate pred);
  template<class ForwardIterator1, class ForwardIterator2>
    ForwardIterator1
      find_end(ForwardIterator1 first1, ForwardIterator1 last1,
               ForwardIterator2 first2, ForwardIterator2 last2);
  template<class ForwardIterator1, class ForwardIterator2,
     class BinaryPredicate>
    ForwardIterator1
      find_end(ForwardIterator1 first1, ForwardIterator1 last1,
```

```
            ForwardIterator2 first2, ForwardIterator2 last2,
            BinaryPredicate pred);

template<class InputIterator, class ForwardIterator>
  InputIterator
    find_first_of(InputIterator first1, InputIterator last1,
                  ForwardIterator first2, ForwardIterator last2);
template<class InputIterator, class ForwardIterator,
   class BinaryPredicate>
  InputIterator
    find_first_of(InputIterator first1, InputIterator last1,
                  ForwardIterator first2, ForwardIterator last2,
                  BinaryPredicate pred);

template<class ForwardIterator>
  ForwardIterator adjacent_find(ForwardIterator first,
                                ForwardIterator last);
template<class ForwardIterator, class BinaryPredicate>
  ForwardIterator adjacent_find(ForwardIterator first,
                                ForwardIterator last,
                                BinaryPredicate pred);

template<class InputIterator, class T>
  typename iterator_traits<InputIterator>::difference_type
    count(InputIterator first, InputIterator last, const T& value);
template<class InputIterator, class Predicate>
  typename iterator_traits<InputIterator>::difference_type
    count_if(InputIterator first, InputIterator last, Predicate pred);

template<class InputIterator1, class InputIterator2>
  pair<InputIterator1, InputIterator2>
    mismatch(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2);
template
 <class InputIterator1, class InputIterator2, class BinaryPredicate>
  pair<InputIterator1, InputIterator2>
    mismatch(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, BinaryPredicate pred);

template<class InputIterator1, class InputIterator2>
  pair<InputIterator1, InputIterator2>
    mismatch(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, InputIterator2 last2);

template
 <class InputIterator1, class InputIterator2, class BinaryPredicate>
  pair<InputIterator1, InputIterator2>
    mismatch(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, InputIterator2 last2,
             BinaryPredicate pred);

template<class InputIterator1, class InputIterator2>
  bool equal(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2);
template
```

```
 <class InputIterator1, class InputIterator2, class BinaryPredicate>
  bool equal(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, BinaryPredicate pred);

template<class InputIterator1, class InputIterator2>
  bool equal(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, InputIterator2 last2);

template
 <class InputIterator1, class InputIterator2, class BinaryPredicate>
  bool equal(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, InputIterator2 last2,
             BinaryPredicate pred);

template<class ForwardIterator1, class ForwardIterator2>
  bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                      ForwardIterator2 first2);
template<class ForwardIterator1, class ForwardIterator2,
                 class BinaryPredicate>
  bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                      ForwardIterator2 first2, BinaryPredicate pred);

template<class ForwardIterator1, class ForwardIterator2>
  bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                      ForwardIterator2 first2, ForwardIterator2 last2);

template<class ForwardIterator1, class ForwardIterator2,
                    class BinaryPredicate>
  bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                      ForwardIterator2 first2, ForwardIterator2 last2,
                      BinaryPredicate pred);

template<class ForwardIterator1, class ForwardIterator2>
  ForwardIterator1 search(
    ForwardIterator1 first1, ForwardIterator1 last1,
    ForwardIterator2 first2, ForwardIterator2 last2);
template<class ForwardIterator1, class ForwardIterator2,
   class BinaryPredicate>
  ForwardIterator1 search(
    ForwardIterator1 first1, ForwardIterator1 last1,
    ForwardIterator2 first2, ForwardIterator2 last2,
    BinaryPredicate pred);
template<class ForwardIterator, class Size, class T>
  ForwardIterator search_n(ForwardIterator first, ForwardIterator last,
                           Size count, const T& value);
template
 <class ForwardIterator, class Size, class T, class BinaryPredicate>
  ForwardIterator search_n(ForwardIterator first, ForwardIterator last,
                           Size count, const T& value,
                           BinaryPredicate pred);

// 25.54, modifying sequence operations:
// 25.54.1, copy:
template<class InputIterator, class OutputIterator>
  OutputIterator copy(InputIterator first, InputIterator last,
```

```
                      OutputIterator result);
template<class InputIterator, class Size, class OutputIterator>
  OutputIterator copy_n(InputIterator first, Size n,
                        OutputIterator result);
template<class InputIterator, class OutputIterator, class Predicate>
  OutputIterator copy_if(InputIterator first, InputIterator last,
                         OutputIterator result, Predicate pred);
template<class BidirectionalIterator1, class BidirectionalIterator2>
  BidirectionalIterator2 copy_backward(
    BidirectionalIterator1 first, BidirectionalIterator1 last,
    BidirectionalIterator2 result);

// 25.65.1, move:
template<class InputIterator, class OutputIterator>
  OutputIterator move(InputIterator first, InputIterator last,
                      OutputIterator result);
template<class BidirectionalIterator1, class BidirectionalIterator2>
  BidirectionalIterator2 move_backward(
    BidirectionalIterator1 first, BidirectionalIterator1 last,
    BidirectionalIterator2 result);

// 25.65.2, swap:
template<class ForwardIterator1, class ForwardIterator2>
  ForwardIterator2 swap_ranges(ForwardIterator1 first1,
  ForwardIterator1 last1, ForwardIterator2 first2);
template<class ForwardIterator1, class ForwardIterator2>
  void iter_swap(ForwardIterator1 a, ForwardIterator2 b);

template<class InputIterator, class OutputIterator, class UnaryOperation>
  OutputIterator transform(InputIterator first, InputIterator last,
                           OutputIterator result, UnaryOperation op);
template<class InputIterator1, class InputIterator2, class OutputIterator,
   class BinaryOperation>
  OutputIterator transform(InputIterator1 first1, InputIterator1 last1,
                           InputIterator2 first2, OutputIterator result,
                           BinaryOperation binary_op);

template<class ForwardIterator, class T>
  void replace(ForwardIterator first, ForwardIterator last,
               const T& old_value, const T& new_value);
template<class ForwardIterator, class Predicate, class T>
  void replace_if(ForwardIterator first, ForwardIterator last,
                  Predicate pred, const T& new_value);
template<class InputIterator, class OutputIterator, class T>
  OutputIterator replace_copy(InputIterator first, InputIterator last,
                              OutputIterator result,
                              const T& old_value, const T& new_value);
template<class InputIterator, class OutputIterator, class Predicate, class T>
  OutputIterator replace_copy_if(InputIterator first, InputIterator last,
                                 OutputIterator result,
                                 Predicate pred, const T& new_value);

template<class ForwardIterator, class T>
  void fill(ForwardIterator first, ForwardIterator last, const T& value);
template<class OutputIterator, class Size, class T>
```

```
  OutputIterator fill_n(OutputIterator first, Size n, const T& value);

template<class ForwardIterator, class Generator>
  void generate(ForwardIterator first, ForwardIterator last,
                Generator gen);
template<class OutputIterator, class Size, class Generator>
  OutputIterator generate_n(OutputIterator first, Size n, Generator gen);

template<class ForwardIterator, class T>
  ForwardIterator remove(ForwardIterator first, ForwardIterator last,
                         const T& value);
template<class ForwardIterator, class Predicate>
  ForwardIterator remove_if(ForwardIterator first, ForwardIterator last,
                            Predicate pred);
template<class InputIterator, class OutputIterator, class T>
  OutputIterator remove_copy(InputIterator first, InputIterator last,
                             OutputIterator result, const T& value);
template<class InputIterator, class OutputIterator, class Predicate>
  OutputIterator remove_copy_if(InputIterator first, InputIterator last,
                                OutputIterator result, Predicate pred);

template<class ForwardIterator>
  ForwardIterator unique(ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class BinaryPredicate>
  ForwardIterator unique(ForwardIterator first, ForwardIterator last,
                         BinaryPredicate pred);
template<class InputIterator, class OutputIterator>
  OutputIterator unique_copy(InputIterator first, InputIterator last,
                             OutputIterator result);
template<class InputIterator, class OutputIterator, class BinaryPredicate>
  OutputIterator unique_copy(InputIterator first, InputIterator last,
                             OutputIterator result, BinaryPredicate pred);

template<class BidirectionalIterator>
  void reverse(BidirectionalIterator first, BidirectionalIterator last);
template<class BidirectionalIterator, class OutputIterator>
  OutputIterator reverse_copy(BidirectionalIterator first,
                              BidirectionalIterator last,
                              OutputIterator result);

template<class ForwardIterator>
  ForwardIterator rotate(ForwardIterator first, ForwardIterator middle,
                         ForwardIterator last);
template<class ForwardIterator, class OutputIterator>
  OutputIterator rotate_copy(
    ForwardIterator first, ForwardIterator middle,
    ForwardIterator last, OutputIterator result);

// D.63, random_shuffle (deprecated):
template<class RandomAccessIterator>
  void random_shuffle(RandomAccessIterator first,
                      RandomAccessIterator last);
template<class RandomAccessIterator, class RandomNumberGenerator>
  void random_shuffle(RandomAccessIterator first,
                      RandomAccessIterator last,
```

```
                      RandomNumberGenerator&& rng);

// 25.96.1, shuffle:
template<class RandomAccessIterator, class UniformRandomNumberGenerator>
  void shuffle(RandomAccessIterator first,
                      RandomAccessIterator last,
                      UniformRandomNumberGenerator&& g);

// 25.96.2, partitions:
template <class InputIterator, class Predicate>
  bool is_partitioned(InputIterator first, InputIterator last, Predicate pred);

template<class ForwardIterator, class Predicate>
  ForwardIterator partition(ForwardIterator first,
                            ForwardIterator last,
                            Predicate pred);
template<class BidirectionalIterator, class Predicate>
  BidirectionalIterator stable_partition(BidirectionalIterator first,
                                         BidirectionalIterator last,
                                         Predicate pred);
template <class InputIterator, class OutputIterator1,
          class OutputIterator2, class Predicate>
  pair<OutputIterator1, OutputIterator2>
  partition_copy(InputIterator first, InputIterator last,
                 OutputIterator1 out_true, OutputIterator2 out_false,
                 Predicate pred);
template<class ForwardIterator, class Predicate>
  ForwardIterator partition_point(ForwardIterator first,
                                  ForwardIterator last,
                                  Predicate pred);

// 25.98, sorting and related operations:
// 25.144.1, sorting:
template<class RandomAccessIterator>
  void sort(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
  void sort(RandomAccessIterator first, RandomAccessIterator last,
            Compare comp);

template<class RandomAccessIterator>
  void stable_sort(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
  void stable_sort(RandomAccessIterator first, RandomAccessIterator last,
                   Compare comp);

template<class RandomAccessIterator>
  void partial_sort(RandomAccessIterator first,
                    RandomAccessIterator middle,
                    RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
  void partial_sort(RandomAccessIterator first,
                    RandomAccessIterator middle,
                    RandomAccessIterator last, Compare comp);
template<class InputIterator, class RandomAccessIterator>
  RandomAccessIterator partial_sort_copy(
```

```
    InputIterator first, InputIterator last,
    RandomAccessIterator result_first,
    RandomAccessIterator result_last);
template<class InputIterator, class RandomAccessIterator, class Compare>
  RandomAccessIterator partial_sort_copy(
    InputIterator first, InputIterator last,
    RandomAccessIterator result_first,
    RandomAccessIterator result_last,
    Compare comp);
template<class ForwardIterator>
  bool is_sorted(ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Compare>
  bool is_sorted(ForwardIterator first, ForwardIterator last,
                 Compare comp);
template<class ForwardIterator>
  ForwardIterator is_sorted_until(ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Compare>
  ForwardIterator is_sorted_until(ForwardIterator first, ForwardIterator last,
                                  Compare comp);

template<class RandomAccessIterator>
  void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                   RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
  void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                   RandomAccessIterator last, Compare comp);

// 25.160.1, binary search:
template<class ForwardIterator, class T>
  ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last,
                              const T& value);
template<class ForwardIterator, class T, class Compare>
  ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last,
                              const T& value, Compare comp);

template<class ForwardIterator, class T>
  ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last,
                              const T& value);
template<class ForwardIterator, class T, class Compare>
  ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last,
                              const T& value, Compare comp);

template<class ForwardIterator, class T>
  pair<ForwardIterator, ForwardIterator>
    equal_range(ForwardIterator first, ForwardIterator last,
                const T& value);
template<class ForwardIterator, class T, class Compare>
  pair<ForwardIterator, ForwardIterator>
    equal_range(ForwardIterator first, ForwardIterator last,
                const T& value, Compare comp);

template<class ForwardIterator, class T>
  bool binary_search(ForwardIterator first, ForwardIterator last,
                     const T& value);
template<class ForwardIterator, class T, class Compare>
```

```
  bool binary_search(ForwardIterator first, ForwardIterator last,
                     const T& value, Compare comp);

// 25.182.1, merge:
template<class InputIterator1, class InputIterator2, class OutputIterator>
  OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
                       InputIterator2 first2, InputIterator2 last2,
                       OutputIterator result);
template<class InputIterator1, class InputIterator2, class OutputIterator,
   class Compare>
  OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
                       InputIterator2 first2, InputIterator2 last2,
                       OutputIterator result, Compare comp);

template<class BidirectionalIterator>
  void inplace_merge(BidirectionalIterator first,
                     BidirectionalIterator middle,
                     BidirectionalIterator last);
template<class BidirectionalIterator, class Compare>
  void inplace_merge(BidirectionalIterator first,
                     BidirectionalIterator middle,
                     BidirectionalIterator last, Compare comp);

// 25.189.1, set operations:
template<class InputIterator1, class InputIterator2>
  bool includes(InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, InputIterator2 last2);
template<class InputIterator1, class InputIterator2, class Compare>
  bool includes(
    InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2, Compare comp);

template<class InputIterator1, class InputIterator2, class OutputIterator>
  OutputIterator set_union(InputIterator1 first1, InputIterator1 last1,
                           InputIterator2 first2, InputIterator2 last2,
                           OutputIterator result);
template<class InputIterator1, class InputIterator2, class OutputIterator,
   class Compare>
  OutputIterator set_union(InputIterator1 first1, InputIterator1 last1,
                           InputIterator2 first2, InputIterator2 last2,
                           OutputIterator result, Compare comp);

template<class InputIterator1, class InputIterator2, class OutputIterator>
  OutputIterator set_intersection(
    InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2,
    OutputIterator result);
template<class InputIterator1, class InputIterator2, class OutputIterator,
   class Compare>
  OutputIterator set_intersection(
    InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2,
    OutputIterator result, Compare comp);

template<class InputIterator1, class InputIterator2, class OutputIterator>
```

```
  OutputIterator set_difference(
    InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2,
    OutputIterator result);
template<class InputIterator1, class InputIterator2, class OutputIterator,
   class Compare>
  OutputIterator set_difference(
    InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2,
    OutputIterator result, Compare comp);

template<class InputIterator1, class InputIterator2, class OutputIterator>
  OutputIterator set_symmetric_difference(
    InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2,
    OutputIterator result);
template<class InputIterator1, class InputIterator2, class OutputIterator,
    class Compare>
  OutputIterator set_symmetric_difference(
    InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2,
    OutputIterator result, Compare comp);

// 25.195.1, heap operations:
template<class RandomAccessIterator>
  void push_heap(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
  void push_heap(RandomAccessIterator first, RandomAccessIterator last,
                 Compare comp);

template<class RandomAccessIterator>
  void pop_heap(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
  void pop_heap(RandomAccessIterator first, RandomAccessIterator last,
                Compare comp);

template<class RandomAccessIterator>
  void make_heap(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
  void make_heap(RandomAccessIterator first, RandomAccessIterator last,
                 Compare comp);

template<class RandomAccessIterator>
  void sort_heap(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
  void sort_heap(RandomAccessIterator first, RandomAccessIterator last,
                 Compare comp);

template<class RandomAccessIterator>
  bool is_heap(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
  bool is_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
template<class RandomAccessIterator>
  RandomAccessIterator is_heap_until(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
```

```
  RandomAccessIterator is_heap_until(RandomAccessIterator first, RandomAccessIterator last,
                                     Compare comp);

// 25.201.1, minimum and maximum:
template<class T> constexpr const T& min(const T& a, const T& b);
template<class T, class Compare>
  constexpr const T& min(const T& a, const T& b, Compare comp);
template<class T>
  constexpr T min(initializer_list<T> t);
template<class T, class Compare>
  constexpr T min(initializer_list<T> t, Compare comp);

template<class T> constexpr const T& max(const T& a, const T& b);
template<class T, class Compare>
  constexpr const T& max(const T& a, const T& b, Compare comp);
template<class T>
  constexpr T max(initializer_list<T> t);
template<class T, class Compare>
  constexpr T max(initializer_list<T> t, Compare comp);

template<class T> constexpr pair<const T&, const T&> minmax(const T& a, const T& b);
template<class T, class Compare>
  constexpr pair<const T&, const T&> minmax(const T& a, const T& b, Compare comp);
template<class T>
  constexpr pair<T, T> minmax(initializer_list<T> t);
template<class T, class Compare>
  constexpr pair<T, T> minmax(initializer_list<T> t, Compare comp);

template<class ForwardIterator>
  ForwardIterator min_element(ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Compare>
  ForwardIterator min_element(ForwardIterator first, ForwardIterator last,
                              Compare comp);
template<class ForwardIterator>
  ForwardIterator max_element(ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Compare>
  ForwardIterator max_element(ForwardIterator first, ForwardIterator last,
                              Compare comp);
template<class ForwardIterator>
  pair<ForwardIterator, ForwardIterator>
    minmax_element(ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Compare>
  pair<ForwardIterator, ForwardIterator>
    minmax_element(ForwardIterator first, ForwardIterator last, Compare comp);

template<class InputIterator1, class InputIterator2>
  bool lexicographical_compare(
    InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2);
template<class InputIterator1, class InputIterator2, class Compare>
  bool lexicographical_compare(
    InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2,
    Compare comp);
```

```
  // 25.204.2, permutations:
  template<class BidirectionalIterator>
    bool next_permutation(BidirectionalIterator first,
                          BidirectionalIterator last);
  template<class BidirectionalIterator, class Compare>
    bool next_permutation(BidirectionalIterator first,
                          BidirectionalIterator last, Compare comp);
  template<class BidirectionalIterator>
    bool prev_permutation(BidirectionalIterator first,
                          BidirectionalIterator last);
  template<class BidirectionalIterator, class Compare>
    bool prev_permutation(BidirectionalIterator first,
                          BidirectionalIterator last, Compare comp);
}
```

3

All of the algorithms are separated from the particular implementations of data structures and are parameterized by iterator types. Because of this, they can work with program-defined data structures, as long as these data structures have iterator types satisfying the assumptions on the algorithms.

1 For purposes of determining the existence of data races, algorithms shall not modify objects referenced through an iterator argument unless the specification requires such modification.

2

Throughout this Clause, the names of template parameters are used to express type requirements.

— If an algorithm's template parameter is named `InputIterator`, `InputIterator1`, or `InputIterator2`, the template argument shall satisfy the requirements of an input iterator (24.100.1).

— If an algorithm's template parameter is named `OutputIterator`, `OutputIterator1`, or `OutputIterator2`, the template argument shall satisfy the requirements of an output iterator (24.116.1).

— If an algorithm's template parameter is named `ForwardIterator`, `ForwardIterator1`, or `ForwardIterator2`, the template argument shall satisfy the requirements of a forward iterator (24.122.1).

— If an algorithm's template parameter is named `BidirectionalIterator`, `BidirectionalIterator1`, or `BidirectionalIterator2`, the template argument shall satisfy the requirements of a bidirectional iterator (24.165.1).

— If an algorithm's template parameter is named `RandomAccessIterator`, `RandomAccessIterator1`, or `RandomAccessIterator2`, the template argument shall satisfy the requirements of a random-access iterator (24.167.1).

1 If an algorithm's **Effects** section says that a value pointed to by any iterator passed as an argument is modified, then that algorithm has an additional type requirement: The type of that argument shall satisfy the requirements of a mutable iterator (24.2). [*Note:* This requirement does not affect arguments that are named `OutputIterator`, `OutputIterator1`, or `OutputIterator2`, because output iterators must always be mutable. — *end note*]

2

Both in-place and copying versions are provided for certain algorithms.[268] When such a version is provided

268) The decision whether to include a copying version was usually based on complexity considerations. When the cost of doing the operation dominates the cost of copy, the copying version is not included. For example, `sort_copy` is not included because the cost of sorting is much more significant, and users might as well do `copy` followed by `sort`.

for *algorithm* it is called *algorithm*`_copy`. Algorithms that take predicates end with the suffix `_if` (which follows the suffix `_copy`).

1

The `Predicate` parameter is used whenever an algorithm expects a function object (20.283) that, when applied to the result of dereferencing the corresponding iterator, returns a value testable as `true`. In other words, if an algorithm takes `Predicate pred` as its argument and `first` as its iterator argument, it should work correctly in the construct `pred(*first)` contextually converted to `bool` (Clause 4). The function object `pred` shall not apply any non-constant function through the dereferenced iterator.

1

The `BinaryPredicate` parameter is used whenever an algorithm expects a function object that when applied to the result of dereferencing two corresponding iterators or to dereferencing an iterator and type `T` when `T` is part of the signature returns a value testable as `true`. In other words, if an algorithm takes `BinaryPredicate binary_pred` as its argument and `first1` and `first2` as its iterator arguments, it should work correctly in the construct `binary_pred(*first1, *first2)` contextually converted to `bool` (Clause 4). `BinaryPredicate` always takes the first iterator's `value_type` as its first argument, that is, in those cases when `T value` is part of the signature, it should work correctly in the construct `binary_pred(*first1, value)` contextually converted to `bool` (Clause 4). `binary_pred` shall not apply any non-constant function through the dereferenced iterators.

1

[*Note:* Unless otherwise specified, algorithms that take function objects as arguments are permitted to copy those function objects freely. Programmers for whom object identity is important should consider using a wrapper class that points to a noncopied implementation object such as `reference_wrapper<T>` (20.339.1), or some equivalent solution. — *end note*]

1 When the description of an algorithm gives an expression such as `*first == value` for a condition, the expression shall evaluate to either true or false in boolean contexts.

2

In the description of the algorithms operators `+` and `-` are used for some of the iterator categories for which they do not have to be defined. In these cases the semantics of `a+n` is the same as that of

```
X tmp = a;
advance(tmp, n);
return tmp;
```

and that of `b-a` is the same as of

```
return distance(a, b);
```

25.23 Non-modifying sequence operations [alg.nonmodifying]

25.23.1 All of [alg.all_of]

```
template <class InputIterator, class Predicate>
  bool all_of(InputIterator first, InputIterator last, Predicate pred);
```

1

Returns: `true` if `[first,last)` is empty or if `pred(*i)` is `true` for every iterator `i` in the range `[first,last)`, and `false` otherwise.

1 *Complexity:* At most `last - first` applications of the predicate.

25.24.1 Any of [alg.any_of]

```
template <class InputIterator, class Predicate>
  bool any_of(InputIterator first, InputIterator last, Predicate pred);
```

1 *Returns:* `false` if `[first,last)` is empty or if there is no iterator `i` in the range `[first,last)` such that `pred(*i)` is `true`, and `true` otherwise.

2 *Complexity:* At most `last - first` applications of the predicate.

25.24.2 None of [alg.none_of]

```
template <class InputIterator, class Predicate>
  bool none_of(InputIterator first, InputIterator last, Predicate pred);
```

1 *Returns:* `true` if `[first,last)` is empty or if `pred(*i)` is `false` for every iterator `i` in the range `[first,last)`, and `false` otherwise.

2 *Complexity:* At most `last - first` applications of the predicate.

25.24.3 For each [alg.foreach]

```
template<class InputIterator, class Function>
  Function for_each(InputIterator first, InputIterator last, Function f);
```

1

Requires: `Function` shall meet the requirements of `MoveConstructible` (Table 20). [*Note:* `Function` need not meet the requirements of `CopyConstructible` (Table 21). — *end note*]

1

Effects: Applies `f` to the result of dereferencing every iterator in the range `[first,last)`, starting from `first` and proceeding to `last - 1`. [*Note:* If the type of `first` satisfies the requirements of a mutable iterator, `f` may apply nonconstant functions through the dereferenced iterator. — *end note*]

1 *Returns:* `std::move(f)`.

2

Complexity: Applies `f` exactly `last - first` times.

1

Remarks: If `f` returns a result, the result is ignored.

25.36.1 Find [alg.find]

```
template<class InputIterator, class T>
  InputIterator find(InputIterator first, InputIterator last,
                     const T& value);

template<class InputIterator, class Predicate>
  InputIterator find_if(InputIterator first, InputIterator last,
                        Predicate pred);
template<class InputIterator, class Predicate>
  InputIterator find_if_not(InputIterator first, InputIterator last,
                            Predicate pred);
```

1

Returns: The first iterator `i` in the range `[first,last)` for which the following corresponding conditions hold: `*i == value`, `pred(*i) != false`, `pred(*i) == false`. Returns `last` if no such iterator is found.

1

Complexity: At most `last - first` applications of the corresponding predicate.

25.42.1 Find end [alg.find.end]

```
template<class ForwardIterator1, class ForwardIterator2>
  ForwardIterator1
    find_end(ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2);

template<class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  ForwardIterator1
    find_end(ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2,
             BinaryPredicate pred);
```

1

Effects: Finds a subsequence of equal values in a sequence.

1 *Returns:* The last iterator `i` in the range `[first1,last1 - (last2 - first2))` such that for every non-negative integer `n < (last2 - first2)`, the following corresponding conditions hold: `*(i + n) == *(first2 + n), pred(*(i + n), *(first2 + n)) != false`. Returns `last1` if `[first2,last2)` is empty or if no such iterator is found.

2

Complexity: At most `(last2 - first2) * (last1 - first1 - (last2 - first2) + 1)` applications of the corresponding predicate.

25.44.1 Find first [alg.find.first.of]

```
template<class InputIterator, class ForwardIterator>
  InputIterator
    find_first_of(InputIterator first1, InputIterator last1,
                  ForwardIterator first2, ForwardIterator last2);

template<class InputIterator, class ForwardIterator,
         class BinaryPredicate>
  InputIterator
    find_first_of(InputIterator first1, InputIterator last1,
                  ForwardIterator first2, ForwardIterator last2,
                  BinaryPredicate pred);
```

1 *Effects:* Finds an element that matches one of a set of values.

2 *Returns:* The first iterator `i` in the range `[first1,last1)` such that for some iterator `j` in the range `[first2,last2)` the following conditions hold: `*i == *j, pred(*i,*j) != false`. Returns `last1` if `[first2,last2)` is empty or if no such iterator is found.

3 *Complexity:* At most `(last1-first1) * (last2-first2)` applications of the corresponding predicate.

25.44.2 Adjacent find [alg.adjacent.find]

```
template<class ForwardIterator>
  ForwardIterator adjacent_find(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class BinaryPredicate>
  ForwardIterator adjacent_find(ForwardIterator first, ForwardIterator last,
                                BinaryPredicate pred);
```

1 *Returns:* The first iterator `i` such that both `i` and `i + 1` are in the range `[first,last)` for which the following corresponding conditions hold: `*i == *(i + 1), pred(*i, *(i + 1)) != false`. Returns `last` if no such iterator is found.

2 *Complexity:* For a nonempty range, exactly `min((i - first) + 1, (last - first) - 1)` applications of the corresponding predicate, where `i` is `adjacent_find`'s return value.

25.44.3 Count [alg.count]

```
template<class InputIterator, class T>
    typename iterator_traits<InputIterator>::difference_type
       count(InputIterator first, InputIterator last, const T& value);

template<class InputIterator, class Predicate>
    typename iterator_traits<InputIterator>::difference_type
      count_if(InputIterator first, InputIterator last, Predicate pred);
```

1 *Effects:* Returns the number of iterators `i` in the range `[first,last)` for which the following corresponding conditions hold: `*i == value, pred(*i) != false`.

2 *Complexity:* Exactly `last - first` applications of the corresponding predicate.

25.45.1 Mismatch [mismatch]

```
template<class InputIterator1, class InputIterator2>
  pair<InputIterator1, InputIterator2>
      mismatch(InputIterator1 first1, InputIterator1 last1,
               InputIterator2 first2);

template<class InputIterator1, class InputIterator2,
          class BinaryPredicate>
  pair<InputIterator1, InputIterator2>
      mismatch(InputIterator1 first1, InputIterator1 last1,
               InputIterator2 first2, BinaryPredicate pred);

template<class InputIterator1, class InputIterator2>
  pair<InputIterator1, InputIterator2>
    mismatch(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, InputIterator2 last2);

template <class InputIterator1, class InputIterator2,
           class BinaryPredicate>
  pair<InputIterator1, InputIterator2>
    mismatch(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, InputIterator2 last2,
             BinaryPredicate pred);
```

1 *Remarks:* If `last2` was not given in the argument list, it denotes `first2 + (last1 - first1)` below.

1 *Returns:* A pair of iterators `i` and `j` such that `j == first2 + (i - first1)` and `i` is the first iterator in the range `[first1,last1)` for which the following corresponding conditions hold:

(1.1) — `j` is in the range `[first2, last2)`.

(1.2) — `!(*i == *(first2 + (i - first1)))`

(1.3) — `pred(*i, *(first2 + (i - first1))) == false`

Returns the pair `first1 + min(last1 - first1, last2 - first2)` and `first2 + min(last1 - first1, last2 - first2)` if such an iterator `i` is not found.

2 *Complexity:* At most `last1 - first1` applications of the corresponding predicate.

25.46.1 Equal [alg.equal]

```
template<class InputIterator1, class InputIterator2>
  bool equal(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2);

template<class InputIterator1, class InputIterator2,
          class BinaryPredicate>
  bool equal(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, BinaryPredicate pred);

template<class InputIterator1, class InputIterator2>
  bool equal(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, InputIterator2 last2);

template<class InputIterator1, class InputIterator2,
           class BinaryPredicate>
  bool equal(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, InputIterator2 last2,
             BinaryPredicate pred);
```

1 *Remarks:* If `last2` was not given in the argument list, it denotes `first2 + (last1 - first1)` below.

2

Returns: If `last1 - first1 != last2 - first2`, return `false`. Otherwise return `true` if for every iterator `i` in the range `[first1,last1)` the following corresponding conditions hold: `*i == *(first2 + (i - first1))`, `pred(*i, *(first2 + (i - first1))) != false`. Otherwise, returns `false`.

1 *Complexity:* No applications of the corresponding predicate if `InputIterator1` and `InputIterator2` meet the requirements of random access iterators and `last1 - first1 != last2 - first2`. Otherwise, at most `min(last1 - first1, last2 - first2)` applications of the corresponding predicate.

25.49.1 Is permutation [alg.is_permutation]

```
template<class ForwardIterator1, class ForwardIterator2>
  bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                      ForwardIterator2 first2);
template<class ForwardIterator1, class ForwardIterator2,
                 class BinaryPredicate>
  bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                      ForwardIterator2 first2, BinaryPredicate pred);
template<class ForwardIterator1, class ForwardIterator2>
  bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                      ForwardIterator2 first2, ForwardIterator2 last2);
template<class ForwardIterator1, class ForwardIterator2,
                 class BinaryPredicate>
  bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                      ForwardIterator2 first2, ForwardIterator2 last2,
                      BinaryPredicate pred);
```

1

Requires: `ForwardIterator1` and `ForwardIterator2` shall have the same value type. The comparison function shall be an equivalence relation.

1 *Remarks:* If `last2` was not given in the argument list, it denotes `first2 + (last1 - first1)` below.

2 *Returns:* If `last1 - first1 != last2 - first2`, return `false`. Otherwise return `true` if there exists a permutation of the elements in the range `[first2,first2 + (last1 - first1))`, beginning with `ForwardIterator2 begin`, such that `equal(first1, last1, begin)` returns `true` or `equal(first1, last1, begin, pred)` returns `true`; otherwise, returns `false`.

3 *Complexity:* No applications of the corresponding predicate if `ForwardIterator1` and `ForwardIterator2` meet the requirements of random access iterators and `last1 - first1 != last2 - first2`. Otherwise, exactly `distance(first1, last1)` applications of the corresponding predicate if `equal(first1, last1, first2, last2)` would return `true` if `pred` was not given in the argument list or `equal(first1, last1, first2, last2, pred)` would return `true` if pred was given in the argument list; otherwise, at worst $\mathcal{O}(N^2)$, where N has the value `distance(first1, last1)`.

25.50.1 Search [alg.search]

```
template<class ForwardIterator1, class ForwardIterator2>
  ForwardIterator1
    search(ForwardIterator1 first1, ForwardIterator1 last1,
           ForwardIterator2 first2, ForwardIterator2 last2);

template<class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  ForwardIterator1
    search(ForwardIterator1 first1, ForwardIterator1 last1,
           ForwardIterator2 first2, ForwardIterator2 last2,
           BinaryPredicate pred);
```

1 *Effects:* Finds a subsequence of equal values in a sequence.

2 *Returns:* The first iterator `i` in the range `[first1,last1 - (last2-first2))` such that for every non-negative integer `n` less than `last2 - first2` the following corresponding conditions hold: `*(i + n) == *(first2 + n), pred(*(i + n), *(first2 + n)) != false`. Returns `first1` if `[first2, last2)` is empty, otherwise returns `last1` if no such iterator is found.

1 *Complexity:* At most `(last1 - first1) * (last2 - first2)` applications of the corresponding predicate.

```
template<class ForwardIterator, class Size, class T>
  ForwardIterator
    search_n(ForwardIterator first, ForwardIterator last, Size count,
             const T& value);

template<class ForwardIterator, class Size, class T,
         class BinaryPredicate>
  ForwardIterator
    search_n(ForwardIterator first, ForwardIterator last, Size count,
             const T& value, BinaryPredicate pred);
```

2 *Requires:* The type `Size` shall be convertible to integral type (4.397, 12.659).

1 *Effects:* Finds a subsequence of equal values in a sequence.

2

Returns: The first iterator `i` in the range `[first,last-count)` such that for every non-negative integer `n` less than `count` the following corresponding conditions hold: `*(i + n) == value, pred(*(i + n),value) != false`. Returns `last` if no such iterator is found.

1 *Complexity:* At most `last - first` applications of the corresponding predicate.

25.54 Mutating sequence operations [alg.modifying.operations]

25.54.1 Copy [alg.copy]

```
template<class InputIterator, class OutputIterator>
  OutputIterator copy(InputIterator first, InputIterator last,
                      OutputIterator result);
```

1

Effects: Copies elements in the range `[first,last)` into the range `[result,result + (last - first))` starting from `first` and proceeding to `last`. For each non-negative integer `n < (last - first)`, performs `*(result + n) = *(first + n)`.

1 *Returns:* `result + (last - first)`.

2 *Requires:* `result` shall not be in the range `[first,last)`.

3

Complexity: Exactly `last - first` assignments.

```
template<class InputIterator, class Size, class OutputIterator>
  OutputIterator copy_n(InputIterator first, Size n,
                        OutputIterator result);
```

1 *Effects:* For each non-negative integer $i < n$, performs `*(result + i) = *(first + i)`.

2 *Returns:* `result + n`.

3 *Complexity:* Exactly `n` assignments.

```
template<class InputIterator, class OutputIterator, class Predicate>
  OutputIterator copy_if(InputIterator first, InputIterator last,
                         OutputIterator result, Predicate pred);
```

4 *Requires:* The ranges `[first,last)` and `[result,result + (last - first))` shall not overlap.

5 *Effects:* Copies all of the elements referred to by the iterator `i` in the range `[first,last)` for which `pred(*i)` is `true`.

6 *Returns:* The end of the resulting range.

7 *Complexity:* Exactly `last - first` applications of the corresponding predicate.

8 *Remarks:* Stable (17.602.0.1).

```
template<class BidirectionalIterator1, class BidirectionalIterator2>
  BidirectionalIterator2
    copy_backward(BidirectionalIterator1 first,
                  BidirectionalIterator1 last,
                  BidirectionalIterator2 result);
```

9 *Effects:* Copies elements in the range `[first,last)` into the range `[result - (last-first),result)` starting from `last - 1` and proceeding to `first`.[269] For each positive integer `n <= (last - first)`, performs `*(result - n) = *(last - n)`.

269) `copy_backward` should be used instead of copy when `last` is in the range `[result - (last - first),result)`.

10 *Requires:* `result` shall not be in the range `(first,last]`.

11 *Returns:* `result - (last - first)`.

12 *Complexity:* Exactly `last - first` assignments.

25.65.1 Move [alg.move]

```
template<class InputIterator, class OutputIterator>
  OutputIterator move(InputIterator first, InputIterator last,
                      OutputIterator result);
```

1 *Effects:* Moves elements in the range `[first,last)` into the range `[result,result + (last - first))` starting from first and proceeding to last. For each non-negative integer `n < (last-first)`, performs `*(result + n) = std::move(*(first + n))`.

2 *Returns:* `result + (last - first)`.

3 *Requires:* `result` shall not be in the range `[first,last)`.

4 *Complexity:* Exactly `last - first` move assignments.

```
template<class BidirectionalIterator1, class BidirectionalIterator2>
  BidirectionalIterator2
    move_backward(BidirectionalIterator1 first,
                  BidirectionalIterator1 last,
                  BidirectionalIterator2 result);
```

5 *Effects:* Moves elements in the range `[first,last)` into the range `[result - (last-first),result )` starting from `last - 1` and proceeding to first.[270] For each positive integer `n <= (last - first)`, performs `*(result - n) = std::move(*(last - n))`.

6 *Requires:* `result` shall not be in the range `(first,last]`.

7 *Returns:* `result - (last - first)`.

8 *Complexity:* Exactly `last - first` assignments.

25.65.2 swap [alg.swap]

```
template<class ForwardIterator1, class ForwardIterator2>
  ForwardIterator2
    swap_ranges(ForwardIterator1 first1, ForwardIterator1 last1,
                ForwardIterator2 first2);
```

1 *Effects:* For each non-negative integer `n < (last1 - first1)` performs: `swap(*(first1 + n), *(first2 + n))`.

2 *Requires:* The two ranges `[first1,last1)` and `[first2,first2 + (last1 - first1))` shall not overlap. `*(first1 + n)` shall be swappable with (17.143.0.1) `*(first2 + n)`.

3 *Returns:* `first2 + (last1 - first1)`.

4 *Complexity:* Exactly `last1 - first1` swaps.

```
template<class ForwardIterator1, class ForwardIterator2>
  void iter_swap(ForwardIterator1 a, ForwardIterator2 b);
```

5 *Effects:* `swap(*a, *b)`.

6 *Requires:* `a` and `b` shall be dereferenceable. `*a` shall be swappable with (17.143.0.1) `*b`.

270) `move_backward` should be used instead of move when last is in the range `[result - (last - first),result)`.

25.65.3 Transform [alg.transform]

```
template<class InputIterator, class OutputIterator,
         class UnaryOperation>
  OutputIterator
    transform(InputIterator first, InputIterator last,
              OutputIterator result, UnaryOperation op);

template<class InputIterator1, class InputIterator2,
         class OutputIterator, class BinaryOperation>
  OutputIterator
    transform(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, OutputIterator result,
              BinaryOperation binary_op);
```

1

Effects: Assigns through every iterator `i` in the range `[result,result + (last1 - first1))` a new corresponding value equal to `op(*(first1 + (i - result))` or `binary_op(*(first1 + (i - result)), *(first2 + (i - result)))`.

1

Requires: `op` and `binary_op` shall not invalidate iterators or subranges, or modify elements in the ranges `[first1,last1]`, `[first2,first2 + (last1 - first1)]`, and `[result,result + (last1 - first1)]`.[271]

1 *Returns:* `result + (last1 - first1)`.

2

Complexity: Exactly `last1 - first1` applications of `op` or `binary_op`.

1

Remarks: `result` may be equal to `first` in case of unary transform, or to `first1` or `first2` in case of binary transform.

25.73.1 Replace [alg.replace]

```
template<class ForwardIterator, class T>
  void replace(ForwardIterator first, ForwardIterator last,
               const T& old_value, const T& new_value);

template<class ForwardIterator, class Predicate, class T>
  void replace_if(ForwardIterator first, ForwardIterator last,
                  Predicate pred, const T& new_value);
```

1 *Requires:* The expression `*first = new_value` shall be valid.

2 *Effects:* Substitutes elements referred by the iterator `i` in the range `[first,last)` with `new_value`, when the following corresponding conditions hold: `*i == old_value`, `pred(*i) != false`.

3 *Complexity:* Exactly `last - first` applications of the corresponding predicate.

```
template<class InputIterator, class OutputIterator, class T>
  OutputIterator
    replace_copy(InputIterator first, InputIterator last,
                 OutputIterator result,
                 const T& old_value, const T& new_value);
```

271) The use of fully closed ranges is intentional.

```
template<class InputIterator, class OutputIterator, class Predicate, class T>
  OutputIterator
    replace_copy_if(InputIterator first, InputIterator last,
                    OutputIterator result,
                    Predicate pred, const T& new_value);
```

4 *Requires:* The results of the expressions `*first` and `new_value` shall be writable to the `result` output iterator. The ranges `[first,last)` and `[result,result + (last - first))` shall not overlap.

5 *Effects:* Assigns to every iterator `i` in the range `[result,result + (last - first))` either `new_value` or `*(first + (i - result))` depending on whether the following corresponding conditions hold:

```
*(first + (i - result)) == old_value
pred(*(first + (i - result))) != false
```

6 *Returns:* `result + (last - first)`.

7 *Complexity:* Exactly `last - first` applications of the corresponding predicate.

25.73.2 Fill [alg.fill]

```
template<class ForwardIterator, class T>
  void fill(ForwardIterator first, ForwardIterator last, const T& value);

template<class OutputIterator, class Size, class T>
  OutputIterator fill_n(OutputIterator first, Size n, const T& value);
```

1 *Requires:* The expression `value` shall be writable to the output iterator. The type `Size` shall be convertible to an integral type (4.397, 12.659).

2 *Effects:* The first algorithm assigns `value` through all the iterators in the range `[first,last)`. The second algorithm assigns `value` through all the iterators in the range `[first,first + n)` if `n` is positive, otherwise it does nothing.

3 *Returns:* `fill_n` returns `first + n` for non-negative values of `n` and `first` for negative values.

4 *Complexity:* Exactly `last - first`, `n`, or 0 assignments, respectively.

25.73.3 Generate [alg.generate]

```
template<class ForwardIterator, class Generator>
  void generate(ForwardIterator first, ForwardIterator last,
                Generator gen);

template<class OutputIterator, class Size, class Generator>
  OutputIterator generate_n(OutputIterator first, Size n, Generator gen);
```

1

Effects: The first algorithm invokes the function object `gen` and assigns the return value of `gen` through all the iterators in the range `[first,last)`. The second algorithm invokes the function object `gen` and assigns the return value of `gen` through all the iterators in the range `[first,first + n)` if `n` is positive, otherwise it does nothing.

1

Requires: `gen` takes no arguments, `Size` shall be convertible to an integral type (4.397, 12.659).

1

Returns: `generate_n` returns `first + n` for non-negative values of `n` and `first` for negative values.

1

Complexity: Exactly `last - first`, `n`, or 0 invocations of `gen` and assignments, respectively.

25.81.1 Remove [alg.remove]

```
template<class ForwardIterator, class T>
  ForwardIterator remove(ForwardIterator first, ForwardIterator last,
                         const T& value);

template<class ForwardIterator, class Predicate>
  ForwardIterator remove_if(ForwardIterator first, ForwardIterator last,
                            Predicate pred);
```

1

Requires: The type of `*first` shall satisfy the `MoveAssignable` requirements (Table 22).

1

Effects: Eliminates all the elements referred to by iterator `i` in the range `[first,last)` for which the following corresponding conditions hold: `*i == value, pred(*i) != false`.

1 *Returns:* The end of the resulting range.

2 *Remarks:* Stable (17.602.0.1).

3 *Complexity:* Exactly `last - first` applications of the corresponding predicate.

4

Note: each element in the range `[ret,last)`, where `ret` is the returned value, has a valid but unspecified state, because the algorithms can eliminate elements by moving from elements that were originally in that range.

```
template<class InputIterator, class OutputIterator, class T>
  OutputIterator
    remove_copy(InputIterator first, InputIterator last,
                OutputIterator result, const T& value);

template<class InputIterator, class OutputIterator, class Predicate>
  OutputIterator
    remove_copy_if(InputIterator first, InputIterator last,
                   OutputIterator result, Predicate pred);
```

1 *Requires:* The ranges `[first,last)` and `[result,result + (last - first))` shall not overlap. The expression `*result = *first` shall be valid.

2 *Effects:* Copies all the elements referred to by the iterator `i` in the range `[first,last)` for which the following corresponding conditions do not hold: `*i == value, pred(*i) != false`.

3 *Returns:* The end of the resulting range.

4 *Complexity:* Exactly `last - first` applications of the corresponding predicate.

5 *Remarks:* Stable (17.602.0.1).

25.90.1 Unique [alg.unique]

```
template<class ForwardIterator>
  ForwardIterator unique(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class BinaryPredicate>
  ForwardIterator unique(ForwardIterator first, ForwardIterator last,
                         BinaryPredicate pred);
```

1

Effects: For a nonempty range, eliminates all but the first element from every consecutive group of equivalent elements referred to by the iterator `i` in the range `[first + 1,last)` for which the following conditions hold: `*(i - 1) == *i` or `pred(*(i - 1), *i) != false`.

1 *Requires:* The comparison function shall be an equivalence relation. The type of `*first` shall satisfy the `MoveAssignable` requirements (Table 22).

2 *Returns:* The end of the resulting range.

3 *Complexity:* For nonempty ranges, exactly `(last - first) - 1` applications of the corresponding predicate.

```
template<class InputIterator, class OutputIterator>
  OutputIterator
    unique_copy(InputIterator first, InputIterator last,
                OutputIterator result);

template<class InputIterator, class OutputIterator,
         class BinaryPredicate>
  OutputIterator
    unique_copy(InputIterator first, InputIterator last,
                OutputIterator result, BinaryPredicate pred);
```

4 *Requires:* The comparison function shall be an equivalence relation. The ranges `[first,last)` and `[result,result+(last-first))` shall not overlap. The expression `*result = *first` shall be valid. If neither `InputIterator` nor `OutputIterator` meets the requirements of forward iterator then the value type of `InputIterator` shall be `CopyConstructible` (Table 21) and `CopyAssignable` (Table 23). Otherwise `CopyConstructible` is not required.

5 *Effects:* Copies only the first element from every consecutive group of equal elements referred to by the iterator `i` in the range `[first,last)` for which the following corresponding conditions hold: `*i == *(i - 1)` or `pred(*i, *(i - 1)) != false`.

6 *Returns:* The end of the resulting range.

7 *Complexity:* For nonempty ranges, exactly `last - first - 1` applications of the corresponding predicate.

25.93.1 Reverse [alg.reverse]

```
template<class BidirectionalIterator>
  void reverse(BidirectionalIterator first, BidirectionalIterator last);
```

1

Effects: For each non-negative integer `i < (last - first)/2`, applies `iter_swap` to all pairs of iterators `first + i, (last - i) - 1`.

1 *Requires:* `*first` shall be swappable (17.143.0.1).

2 *Complexity:* Exactly `(last - first)/2` swaps.

```
template<class BidirectionalIterator, class OutputIterator>
  OutputIterator
    reverse_copy(BidirectionalIterator first,
                 BidirectionalIterator last, OutputIterator result);
```

3

Effects: Copies the range `[first,last)` to the range `[result,result+(last-first))` such that for every non-negative integer `i < (last - first)` the following assignment takes place: `*(result + (last - first) - 1 - i) = *(first + i)`.

1 *Requires:* The ranges `[first,last)` and `[result,result+(last-first))` shall not overlap.

2 *Returns:* `result + (last - first)`.

3 *Complexity:* Exactly `last - first` assignments.

25.95.1 Rotate [alg.rotate]

```
template<class ForwardIterator>
  ForwardIterator rotate(ForwardIterator first, ForwardIterator middle,
              ForwardIterator last);
```

1 *Effects:* For each non-negative integer `i < (last - first)`, places the element from the position `first + i` into position `first + (i + (last - middle)) % (last - first)`.

2 *Returns:* `first + (last - middle)`.

3 *Remarks:* This is a left rotate.

4 *Requires:* `[first,middle)` and `[middle,last)` shall be valid ranges. `ForwardIterator` shall satisfy the requirements of `ValueSwappable` (17.143.0.1). The type of `*first` shall satisfy the requirements of `MoveConstructible` (Table 20) and the requirements of `MoveAssignable` (Table 22).

5

Complexity: At most `last - first` swaps.

```
template<class ForwardIterator, class OutputIterator>
  OutputIterator
    rotate_copy(ForwardIterator first, ForwardIterator middle,
                ForwardIterator last, OutputIterator result);
```

1 *Effects:* Copies the range `[first,last)` to the range `[result,result + (last - first))` such that for each non-negative integer `i < (last - first)` the following assignment takes place: `*(result + i) = *(first + (i + (middle - first)) % (last - first))`.

2 *Returns:* `result + (last - first)`.

3 *Requires:* The ranges `[first,last)` and `[result,result + (last - first))` shall not overlap.

4 *Complexity:* Exactly `last - first` assignments.

25.96.1 Shuffle [alg.random.shuffle]

```
template<class RandomAccessIterator, class UniformRandomNumberGenerator>
  void shuffle(RandomAccessIterator first,
                      RandomAccessIterator last,
                      UniformRandomNumberGenerator&& g);
```

1 *Effects:* Permutes the elements in the range `[first,last)` such that each possible permutation of those elements has equal probability of appearance.

2 *Requires:* `RandomAccessIterator` shall satisfy the requirements of `ValueSwappable` (17.143.0.1). The type `UniformRandomNumberGenerator` shall meet the requirements of a uniform random number generator (26.75.0.1) type whose return type is convertible to `iterator_traits<RandomAccessIterator>::difference_type`.

3 *Complexity:* Exactly `(last - first) - 1` swaps.

4 *Remarks:* To the extent that the implementation of this function makes use of random numbers, the object `g` shall serve as the implementation's source of randomness.

25.96.2 Partitions [alg.partitions]

```
template <class InputIterator, class Predicate>
  bool is_partitioned(InputIterator first, InputIterator last, Predicate pred);
```

1 *Requires:* `InputIterator`'s value type shall be convertible to `Predicate`'s argument type.

2 *Returns:* `true` if `[first,last)` is empty or if `[first,last)` is partitioned by `pred`, i.e. if all elements that satisfy `pred` appear before those that do not.

3 *Complexity:* Linear. At most `last - first` applications of `pred`.

```
template<class ForwardIterator, class Predicate>
  ForwardIterator
    partition(ForwardIterator first,
              ForwardIterator last, Predicate pred);
```

4 *Effects:* Places all the elements in the range `[first,last)` that satisfy `pred` before all the elements that do not satisfy it.

5 *Returns:* An iterator `i` such that for every iterator `j` in the range `[first,i) pred(*j) != false`, and for every iterator `k` in the range `[i,last)`, `pred(*k) == false`.

1 *Requires:* `ForwardIterator` shall satisfy the requirements of `ValueSwappable` (17.143.0.1).

2 *Complexity:* If ForwardIterator meets the requirements for a BidirectionalIterator, at most `(last - first) / 2` swaps are done; otherwise at most `last - first` swaps are done. Exactly `last - first` applications of the predicate are done.

```
template<class BidirectionalIterator, class Predicate>
  BidirectionalIterator
    stable_partition(BidirectionalIterator first,
                     BidirectionalIterator last, Predicate pred);
```

3 *Effects:* Places all the elements in the range `[first,last)` that satisfy `pred` before all the elements that do not satisfy it.

4 *Returns:* An iterator `i` such that for every iterator `j` in the range `[first,i)`, `pred(*j) != false`, and for every iterator `k` in the range `[i,last)`, `pred(*k) == false`. The relative order of the elements in both groups is preserved.

5 *Requires:* `BidirectionalIterator` shall satisfy the requirements of `ValueSwappable` (17.143.0.1). The type of `*first` shall satisfy the requirements of `MoveConstructible` (Table 20) and of `MoveAssignable` (Table 22).

6 *Complexity:* At most `(last - first) * log(last - first)` swaps, but only linear number of swaps if there is enough extra memory. Exactly `last - first` applications of the predicate.

```
template <class InputIterator, class OutputIterator1,
          class OutputIterator2, class Predicate>
  pair<OutputIterator1, OutputIterator2>
  partition_copy(InputIterator first, InputIterator last,
                 OutputIterator1 out_true, OutputIterator2 out_false,
                 Predicate pred);
```

7 *Requires:* `InputIterator`'s value type shall be `CopyAssignable`, and shall be writable to the `out_true` and `out_false` `OutputIterator`s, and shall be convertible to `Predicate`'s argument type. The input range shall not overlap with either of the output ranges.

8 *Effects:* For each iterator `i` in `[first,last)`, copies `*i` to the output range beginning with `out_true` if `pred(*i)` is `true`, or to the output range beginning with `out_false` otherwise.

9 *Returns:* A pair `p` such that `p.first` is the end of the output range beginning at `out_true` and `p.second` is the end of the output range beginning at `out_false`.

10 *Complexity:* Exactly `last - first` applications of `pred`.

```
template<class ForwardIterator, class Predicate>
  ForwardIterator partition_point(ForwardIterator first,
                                  ForwardIterator last,
                                  Predicate pred);
```

11 *Requires:* `ForwardIterator`'s value type shall be convertible to `Predicate`'s argument type. `[first, last)` shall be partitioned by `pred`, i.e. all elements that satisfy `pred` shall appear before those that do not.

12 *Returns:* An iterator `mid` such that `all_of(first, mid, pred)` and `none_of(mid, last, pred)` are both true.

13 *Complexity:* $\mathcal{O}(log(last - first))$ applications of `pred`.

25.98 Sorting and related operations [alg.sorting]

1

All the operations in 25.98 have two versions: one that takes a function object of type `Compare` and one that uses an `operator<`.

1

`Compare` is a function object type (20.283). The return value of the function call operation applied to an object of type `Compare`, when contextually converted to `bool` (Clause 4), yields `true` if the first argument of the call is less than the second, and `false` otherwise. `Compare comp` is used throughout for algorithms assuming an ordering relation. It is assumed that `comp` will not apply any non-constant function through the dereferenced iterator.

1

For all algorithms that take `Compare`, there is a version that uses `operator<` instead. That is, `comp(*i, *j) != false` defaults to `*i < *j != false`. For algorithms other than those described in 25.160.1 to work correctly, `comp` has to induce a strict weak ordering on the values.

1

The term *strict* refers to the requirement of an irreflexive relation (`!comp(x, x)` for all `x`), and the term *weak* to requirements that are not as strong as those for a total ordering, but stronger than those for a partial ordering. If we define `equiv(a, b)` as `!comp(a, b) && !comp(b, a)`, then the requirements are that `comp` and `equiv` both be transitive relations:

— `comp(a, b) && comp(b, c)` implies `comp(a, c)`

— `equiv(a, b) && equiv(b, c)` implies `equiv(a, c)` [*Note:* Under these conditions, it can be shown that

  — `equiv` is an equivalence relation

  — `comp` induces a well-defined relation on the equivalence classes determined by `equiv`

  — The induced relation is a strict total ordering. — *end note*]

1 A sequence is *sorted with respect to a comparator* `comp` if for every iterator `i` pointing to the sequence and every non-negative integer `n` such that `i + n` is a valid iterator pointing to an element of the sequence, `comp(*(i + n), *i) == false`.

2 A sequence `[start,finish)` is *partitioned with respect to an expression* `f(e)` if there exists an integer `n` such that for all `0 <= i < distance(start, finish)`, `f(*(start + i))` is true if and only if `i < n`.

3 In the descriptions of the functions that deal with ordering relationships we frequently use a notion of equivalence to describe concepts such as stability. The equivalence to which we refer is not necessarily an `operator==`, but an equivalence relation induced by the strict weak ordering. That is, two elements `a` and `b` are considered equivalent if and only if `!(a < b) && !(b < a)`.

25.144.1 Sorting [alg.sort]

25.144.1.1 `sort` [sort]

```
template<class RandomAccessIterator>
  void sort(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  void sort(RandomAccessIterator first, RandomAccessIterator last,
            Compare comp);
```

1 *Effects:* Sorts the elements in the range `[first,last)`.

2 *Requires:* `RandomAccessIterator` shall satisfy the requirements of `ValueSwappable` (17.143.0.1). The type of `*first` shall satisfy the requirements of `MoveConstructible` (Table 20) and of `MoveAssignable` (Table 22).

1 *Complexity:* $\mathcal{O}(N \log(N))$ (where N `== last - first`) comparisons.

25.154.0.1 `stable_sort` [stable.sort]

```
template<class RandomAccessIterator>
  void stable_sort(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  void stable_sort(RandomAccessIterator first, RandomAccessIterator last,
                   Compare comp);
```

1 *Effects:* Sorts the elements in the range `[first,last)`.

2 *Requires:* `RandomAccessIterator` shall satisfy the requirements of `ValueSwappable` (17.143.0.1). The type of `*first` shall satisfy the requirements of `MoveConstructible` (Table 20) and of `MoveAssignable` (Table 22).

3 *Complexity:* It does at most $N \log^2(N)$ (where N `== last - first`) comparisons; if enough extra memory is available, it is $N \log(N)$.

1 *Remarks:* Stable (17.602.0.1).

25.156.0.1 `partial_sort` [partial.sort]

```
template<class RandomAccessIterator>
  void partial_sort(RandomAccessIterator first,
                    RandomAccessIterator middle,
                    RandomAccessIterator last);
```

```
template<class RandomAccessIterator, class Compare>
  void partial_sort(RandomAccessIterator first,
                    RandomAccessIterator middle,
                    RandomAccessIterator last,
                    Compare comp);
```

1

Effects: Places the first `middle - first` sorted elements from the range `[first,last)` into the range `[first,middle)`. The rest of the elements in the range `[middle,last)` are placed in an unspecified order.

1 *Requires:* `RandomAccessIterator` shall satisfy the requirements of `ValueSwappable` (17.143.0.1). The type of `*first` shall satisfy the requirements of `MoveConstructible` (Table 20) and of `MoveAssignable` (Table 22).

2

Complexity: It takes approximately `(last - first) * log(middle - first)` comparisons.

25.158.0.1 `partial_sort_copy` [partial.sort.copy]

```
template<class InputIterator, class RandomAccessIterator>
  RandomAccessIterator
    partial_sort_copy(InputIterator first, InputIterator last,
                      RandomAccessIterator result_first,
                      RandomAccessIterator result_last);

template<class InputIterator, class RandomAccessIterator,
         class Compare>
  RandomAccessIterator
    partial_sort_copy(InputIterator first, InputIterator last,
                      RandomAccessIterator result_first,
                      RandomAccessIterator result_last,
                      Compare comp);
```

1 *Effects:* Places the first `min(last - first, result_last - result_first)` sorted elements into the range `[result_first,result_first + min(last - first, result_last - result_first))`.

2 *Returns:* The smaller of: `result_last` or `result_first + (last - first)`.

3 *Requires:* `RandomAccessIterator` shall satisfy the requirements of `ValueSwappable` (17.143.0.1). The type of `*result_first` shall satisfy the requirements of `MoveConstructible` (Table 20) and of `MoveAssignable` (Table 22).

4 *Complexity:* Approximately `(last - first) * log(min(last - first, result_last - result_first))` comparisons.

25.158.0.2 `is_sorted` [is.sorted]

```
template<class ForwardIterator>
  bool is_sorted(ForwardIterator first, ForwardIterator last);
```

1 *Returns:* `is_sorted_until(first, last) == last`

```
template<class ForwardIterator, class Compare>
  bool is_sorted(ForwardIterator first, ForwardIterator last,
    Compare comp);
```

2 *Returns:* `is_sorted_until(first, last, comp) == last`

```
template<class ForwardIterator>
  ForwardIterator is_sorted_until(ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Compare>
  ForwardIterator is_sorted_until(ForwardIterator first, ForwardIterator last,
    Compare comp);
```

3 *Returns:* If `distance(first, last) < 2`, returns `last`. Otherwise, returns the last iterator `i` in `[first,last]` for which the range `[first,i)` is sorted.

4 *Complexity:* Linear.

25.158.1 Nth element [alg.nth.element]

```
template<class RandomAccessIterator>
  void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                   RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                   RandomAccessIterator last,  Compare comp);
```

1

After `nth_element` the element in the position pointed to by `nth` is the element that would be in that position if the whole range were sorted, unless `nth == last`. Also for every iterator `i` in the range `[first,nth)` and every iterator `j` in the range `[nth,last)` it holds that: `!(*j < *i)` or `comp(*j, *i) == false`.

1 *Requires:* `RandomAccessIterator` shall satisfy the requirements of `ValueSwappable` (17.143.0.1). The type of `*first` shall satisfy the requirements of `MoveConstructible` (Table 20) and of `MoveAssignable` (Table 22).

2 *Complexity:* Linear on average.

25.160.1 Binary search [alg.binary.search]

1

All of the algorithms in this section are versions of binary search and assume that the sequence being searched is partitioned with respect to an expression formed by binding the search key to an argument of the implied or explicit comparison function. They work on non-random access iterators minimizing the number of comparisons, which will be logarithmic for all types of iterators. They are especially appropriate for random access iterators, because these algorithms do a logarithmic number of steps through the data structure. For non-random access iterators they execute a linear number of steps.

25.163.0.1 `lower_bound` [lower.bound]

```
template<class ForwardIterator, class T>
  ForwardIterator
    lower_bound(ForwardIterator first, ForwardIterator last,
                const T& value);

template<class ForwardIterator, class T, class Compare>
  ForwardIterator
    lower_bound(ForwardIterator first, ForwardIterator last,
                const T& value, Compare comp);
```

1

Requires: The elements `e` of `[first,last)` shall be partitioned with respect to the expression `e < value` or `comp(e, value)`.

1

Returns: The furthermost iterator `i` in the range `[first,last]` such that for every iterator `j` in the range `[first,i)` the following corresponding conditions hold: `*j < value` or `comp(*j, value) != false`.

1 *Complexity:* At most $\log_2$(`last - first`) + $\mathcal{O}(1)$ comparisons.

25.173.0.1 `upper_bound` [upper.bound]

```
template<class ForwardIterator, class T>
  ForwardIterator
    upper_bound(ForwardIterator first, ForwardIterator last,
                const T& value);

template<class ForwardIterator, class T, class Compare>
  ForwardIterator
    upper_bound(ForwardIterator first, ForwardIterator last,
                const T& value, Compare comp);
```

1

Requires: The elements `e` of `[first,last)` shall be partitioned with respect to the expression `!(value < e)` or `!comp(value, e)`.

1

Returns: The furthermost iterator `i` in the range `[first,last]` such that for every iterator `j` in the range `[first,i)` the following corresponding conditions hold: `!(value < *j)` or `comp(value, *j) == false`.

1 *Complexity:* At most $\log_2$(`last - first`) + $\mathcal{O}(1)$ comparisons.

25.178.0.1 `equal_range` [equal.range]

```
template<class ForwardIterator, class T>
  pair<ForwardIterator, ForwardIterator>
    equal_range(ForwardIterator first,
                ForwardIterator last, const T& value);

template<class ForwardIterator, class T, class Compare>
  pair<ForwardIterator, ForwardIterator>
    equal_range(ForwardIterator first,
                ForwardIterator last, const T& value,
                Compare comp);
```

1

Requires: The elements `e` of `[first,last)` shall be partitioned with respect to the expressions `e < value` and `!(value < e)` or `comp(e, value)` and `!comp(value, e)`. Also, for all elements `e` of `[first, last)`, `e < value` shall imply `!(value < e)` or `comp(e, value)` shall imply `!comp(value, e)`.

1 *Returns:*

```
make_pair(lower_bound(first, last, value),
          upper_bound(first, last, value))
```

or

```
make_pair(lower_bound(first, last, value, comp),
          upper_bound(first, last, value, comp))
```

2 *Complexity:* At most $2 * \log_2$(`last - first`) + $\mathcal{O}(1)$ comparisons.

25.179.0.1 `binary_search` [binary.search]

```
template<class ForwardIterator, class T>
  bool binary_search(ForwardIterator first, ForwardIterator last,
                     const T& value);

template<class ForwardIterator, class T, class Compare>
  bool binary_search(ForwardIterator first, ForwardIterator last,
                     const T& value, Compare comp);
```

1 *Requires:* The elements `e` of `[first,last)` are partitioned with respect to the expressions `e < value` and `!(value < e)` or `comp(e, value)` and `!comp(value, e)`. Also, for all elements `e` of `[first, last)`, `e < value` implies `!(value < e)` or `comp(e, value)` implies `!comp(value, e)`.

1 *Returns:* `true` if there is an iterator `i` in the range `[first,last)` that satisfies the corresponding conditions: `!(*i < value) && !(value < *i)` or `comp(*i, value) == false && comp(value, *i) == false`.

1 *Complexity:* At most $\log_2$(`last - first`) + $\mathcal{O}(1)$ comparisons.

25.182.1 Merge [alg.merge]

```
template<class InputIterator1, class InputIterator2,
         class OutputIterator>
  OutputIterator
    merge(InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2, InputIterator2 last2,
          OutputIterator result);

template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
  OutputIterator
    merge(InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2, InputIterator2 last2,
          OutputIterator result, Compare comp);
```

1 *Effects:* Copies all the elements of the two ranges `[first1,last1)` and `[first2,last2)` into the range `[result,result_last)`, where `result_last` is `result + (last1 - first1) + (last2 - first2)`, such that the resulting range satisfies `is_sorted(result, result_last)` or `is_sorted(result, result_last, comp)`, respectively.

2 *Requires:* The ranges `[first1,last1)` and `[first2,last2)` shall be sorted with respect to `operator<` or `comp`. The resulting range shall not overlap with either of the original ranges.

1 *Returns:* `result + (last1 - first1) + (last2 - first2)`.

2 *Complexity:* At most `(last1 - first1) + (last2 - first2) - 1` comparisons.

1 *Remarks:* Stable (17.602.0.1).

```
template<class BidirectionalIterator>
  void inplace_merge(BidirectionalIterator first,
                     BidirectionalIterator middle,
                     BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
  void inplace_merge(BidirectionalIterator first,
                     BidirectionalIterator middle,
                     BidirectionalIterator last, Compare comp);
```

1 *Effects:* Merges two sorted consecutive ranges `[first,middle)` and `[middle,last)`, putting the result of the merge into the range `[first,last)`. The resulting range will be in non-decreasing order; that is, for every iterator `i` in `[first,last)` other than `first`, the condition `*i < *(i - 1)` or, respectively, `comp(*i, *(i - 1))` will be false.

2 *Requires:* The ranges `[first,middle)` and `[middle,last)` shall be sorted with respect to `operator<` or `comp`. `BidirectionalIterator` shall satisfy the requirements of `ValueSwappable` (17.143.0.1). The type of `*first` shall satisfy the requirements of `MoveConstructible` (Table 20) and of `MoveAssignable` (Table 22).

3 *Complexity:* When enough additional memory is available, `(last - first) - 1` comparisons. If no additional memory is available, an algorithm with complexity $N \log(N)$ (where `N` is equal to `last - first`) may be used.

1 *Remarks:* Stable (17.602.0.1).

25.189.1 Set operations on sorted structures [alg.set.operations]

1 This section defines all the basic set operations on sorted structures. They also work with `multisets` (23.931.1) containing multiple copies of equivalent elements. The semantics of the set operations are generalized to `multisets` in a standard way by defining `set_union()` to contain the maximum number of occurrences of every element, `set_intersection()` to contain the minimum, and so on.

25.194.0.1 `includes` [includes]

```
template<class InputIterator1, class InputIterator2>
  bool includes(InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, InputIterator2 last2);

template<class InputIterator1, class InputIterator2, class Compare>
  bool includes(InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, InputIterator2 last2,
                Compare comp);
```

1 *Returns:* `true` if `[first2,last2)` is empty or if every element in the range `[first2,last2)` is contained in the range `[first1,last1)`. Returns `false` otherwise.

2 *Complexity:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` comparisons.

25.194.0.2 `set_union` [set.union]

```
template<class InputIterator1, class InputIterator2,
         class OutputIterator>
  OutputIterator
    set_union(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2,
```

```
              OutputIterator result);

template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
  OutputIterator
    set_union(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2,
              OutputIterator result, Compare comp);
```

1 *Effects:* Constructs a sorted union of the elements from the two ranges; that is, the set of elements that are present in one or both of the ranges.

2 *Requires:* The resulting range shall not overlap with either of the original ranges.

3 *Returns:* The end of the constructed range.

4 *Complexity:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` comparisons.

5 *Remarks:* If `[first1,last1)` contains m elements that are equivalent to each other and `[first2, last2)` contains n elements that are equivalent to them, then all m elements from the first range shall be copied to the output range, in order, and then $\max(n - m, 0)$ elements from the second range shall be copied to the output range, in order.

25.194.0.3 `set_intersection` [set.intersection]

```
template<class InputIterator1, class InputIterator2,
         class OutputIterator>
  OutputIterator
    set_intersection(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result);

template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
  OutputIterator
    set_intersection(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result, Compare comp);
```

1 *Effects:* Constructs a sorted intersection of the elements from the two ranges; that is, the set of elements that are present in both of the ranges.

2 *Requires:* The resulting range shall not overlap with either of the original ranges.

3 *Returns:* The end of the constructed range.

4 *Complexity:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` comparisons.

5 *Remarks:* If `[first1,last1)` contains m elements that are equivalent to each other and `[first2, last2)` contains n elements that are equivalent to them, the first $\min(m, n)$ elements shall be copied from the first range to the output range, in order.

25.195.0.1 `set_difference` [set.difference]

```
template<class InputIterator1, class InputIterator2,
         class OutputIterator>
  OutputIterator
    set_difference(InputIterator1 first1, InputIterator1 last1,
                   InputIterator2 first2, InputIterator2 last2,
```

```
                    OutputIterator result);

template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
  OutputIterator
    set_difference(InputIterator1 first1, InputIterator1 last1,
                   InputIterator2 first2, InputIterator2 last2,
                   OutputIterator result, Compare comp);
```

1 *Effects:* Copies the elements of the range `[first1,last1)` which are not present in the range `[first2, last2)` to the range beginning at `result`. The elements in the constructed range are sorted.

2 *Requires:* The resulting range shall not overlap with either of the original ranges.

3 *Returns:* The end of the constructed range.

4 *Complexity:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` comparisons.

5 *Remarks:* If `[first1,last1)` contains m elements that are equivalent to each other and `[first2, last2)` contains n elements that are equivalent to them, the last $\max(m-n, 0)$ elements from `[first1, last1)` shall be copied to the output range.

25.195.0.2 `set_symmetric_difference` [set.symmetric.difference]

```
template<class InputIterator1, class InputIterator2,
         class OutputIterator>
  OutputIterator
    set_symmetric_difference(InputIterator1 first1, InputIterator1 last1,
                             InputIterator2 first2, InputIterator2 last2,
                             OutputIterator result);

template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
  OutputIterator
    set_symmetric_difference(InputIterator1 first1, InputIterator1 last1,
                             InputIterator2 first2, InputIterator2 last2,
                             OutputIterator result, Compare comp);
```

1 *Effects:* Copies the elements of the range `[first1,last1)` that are not present in the range `[first2, last2)`, and the elements of the range `[first2,last2)` that are not present in the range `[first1, last1)` to the range beginning at `result`. The elements in the constructed range are sorted.

2 *Requires:* The resulting range shall not overlap with either of the original ranges.

3 *Returns:* The end of the constructed range.

4 *Complexity:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` comparisons.

5 *Remarks:* If `[first1,last1)` contains m elements that are equivalent to each other and `[first2, last2)` contains n elements that are equivalent to them, then $|m-n|$ of those elements shall be copied to the output range: the last $m-n$ of these elements from `[first1,last1)` if $m > n$, and the last $n-m$ of these elements from `[first2,last2)` if $m < n$.

25.195.1 Heap operations [alg.heap.operations]

1

A *heap* is a particular organization of elements in a range between two random access iterators `[a,b)`. Its two key properties are:

(1) There is no element greater than `*a` in the range and

(2) `*a` may be removed by `pop_heap()`, or a new element added by `push_heap()`, in $\mathcal{O}(\log(N))$ time.

1 These properties make heaps useful as priority queues.

2 `make_heap()` converts a range into a heap and `sort_heap()` turns a heap into a sorted sequence.

25.196.0.1 `push_heap` [push.heap]

```
template<class RandomAccessIterator>
  void push_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  void push_heap(RandomAccessIterator first, RandomAccessIterator last,
                 Compare comp);
```

1 *Effects:* Places the value in the location `last - 1` into the resulting heap `[first,last)`.

2 *Requires:* The range `[first,last - 1)` shall be a valid heap. The type of `*first` shall satisfy the `MoveConstructible` requirements (Table 20) and the `MoveAssignable` requirements (Table 22).

3 *Complexity:* At most `log(last - first)` comparisons.

25.198.0.1 `pop_heap` [pop.heap]

```
template<class RandomAccessIterator>
  void pop_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  void pop_heap(RandomAccessIterator first, RandomAccessIterator last,
                Compare comp);
```

1 *Requires:* The range `[first,last)` shall be a valid non-empty heap. `RandomAccessIterator` shall satisfy the requirements of `ValueSwappable` (17.143.0.1). The type of `*first` shall satisfy the requirements of `MoveConstructible` (Table 20) and of `MoveAssignable` (Table 22).

2 *Effects:* Swaps the value in the location `first` with the value in the location `last - 1` and makes `[first,last - 1)` into a heap.

1 *Complexity:* At most `2 * log(last - first)` comparisons.

25.199.0.1 `make_heap` [make.heap]

```
template<class RandomAccessIterator>
  void make_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  void make_heap(RandomAccessIterator first, RandomAccessIterator last,
                 Compare comp);
```

1 *Effects:* Constructs a heap out of the range `[first,last)`.

2 *Requires:* The type of `*first` shall satisfy the `MoveConstructible` requirements (Table 20) and the `MoveAssignable` requirements (Table 22).

3 *Complexity:* At most `3 * (last - first)` comparisons.

25.199.0.2 `sort_heap` [sort.heap]

```
template<class RandomAccessIterator>
  void sort_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  void sort_heap(RandomAccessIterator first, RandomAccessIterator last,
                 Compare comp);
```

1 *Effects:* Sorts elements in the heap `[first,last)`.

2 *Requires:* The range `[first,last)` shall be a valid heap. `RandomAccessIterator` shall satisfy the requirements of `ValueSwappable` (17.143.0.1). The type of `*first` shall satisfy the requirements of `MoveConstructible` (Table 20) and of `MoveAssignable` (Table 22).

3 *Complexity:* At most $N \log(N)$ comparisons (where `N == last - first`).

25.199.0.3 `is_heap` [is.heap]

```
template<class RandomAccessIterator>
  bool is_heap(RandomAccessIterator first, RandomAccessIterator last);
```

1 *Returns:* `is_heap_until(first, last) == last`

```
template<class RandomAccessIterator, class Compare>
  bool is_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

1 *Returns:* `is_heap_until(first, last, comp) == last`

```
template<class RandomAccessIterator>
  RandomAccessIterator is_heap_until(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
  RandomAccessIterator is_heap_until(RandomAccessIterator first, RandomAccessIterator last,
    Compare comp);
```

2 *Returns:* If `distance(first, last) < 2`, returns `last`. Otherwise, returns the last iterator `i` in `[first,last]` for which the range `[first,i)` is a heap.

3 *Complexity:* Linear.

25.201.1 Minimum and maximum [alg.min.max]

```
template<class T> constexpr const T& min(const T& a, const T& b);
template<class T, class Compare>
  constexpr const T& min(const T& a, const T& b, Compare comp);
```

1 *Requires:* Type `T` is `LessThanComparable` (Table 18).

2 *Returns:* The smaller value.

3 *Remarks:* Returns the first argument when the arguments are equivalent.

```
template<class T>
  constexpr T min(initializer_list<T> t);
template<class T, class Compare>
  constexpr T min(initializer_list<T> t, Compare comp);
```

1 *Requires:* `T` is `LessThanComparable` and `CopyConstructible` and `t.size() > 0`.

2 *Returns:* The smallest value in the initializer_list.

3 *Remarks:* Returns a copy of the leftmost argument when several arguments are equivalent to the smallest.

```
template<class T> constexpr const T& max(const T& a, const T& b);
template<class T, class Compare>
  constexpr const T& max(const T& a, const T& b, Compare comp);
```

1 *Requires:* Type `T` is `LessThanComparable` (Table 18).

2 *Returns:* The larger value.

3 *Remarks:* Returns the first argument when the arguments are equivalent.

```
template<class T>
  constexpr T max(initializer_list<T> t);
template<class T, class Compare>
  constexpr T max(initializer_list<T> t, Compare comp);
```

4 *Requires:* `T` is `LessThanComparable` and `CopyConstructible` and `t.size() > 0`.

5 *Returns:* The largest value in the initializer_list.

6 *Remarks:* Returns a copy of the leftmost argument when several arguments are equivalent to the largest.

```
template<class T> constexpr pair<const T&, const T&> minmax(const T& a, const T& b);
template<class T, class Compare>
  constexpr pair<const T&, const T&> minmax(const T& a, const T& b, Compare comp);
```

7 *Requires:* Type `T` shall be `LessThanComparable` (Table 18).

8 *Returns:* `pair<const T&, const T&>(b, a)` if `b` is smaller than `a`, and `pair<const T&, const T&>(a, b)` otherwise.

9 *Remarks:* Returns `pair<const T&, const T&>(a, b)` when the arguments are equivalent.

10 *Complexity:* Exactly one comparison.

```
template<class T>
  constexpr pair<T, T> minmax(initializer_list<T> t);
template<class T, class Compare>
  constexpr pair<T, T> minmax(initializer_list<T> t, Compare comp);
```

11 *Requires:* `T` is `LessThanComparable` and `CopyConstructible` and `t.size() > 0`.

12 *Returns:* `pair<T, T>(x, y)`, where `x` has the smallest and `y` has the largest value in the initializer list.

13 *Remarks:* `x` is a copy of the leftmost argument when several arguments are equivalent to the smallest. `y` is a copy of the rightmost argument when several arguments are equivalent to the largest.

14 *Complexity:* At most `(3/2) * t.size()` applications of the corresponding predicate.

```
template<class ForwardIterator>
  ForwardIterator min_element(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
  ForwardIterator min_element(ForwardIterator first, ForwardIterator last,
                              Compare comp);
```

15

Returns: The first iterator `i` in the range `[first,last)` such that for every iterator `j` in the range `[first,last)` the following corresponding conditions hold: `!(*j < *i)` or `comp(*j, *i) == false`. Returns `last` if `first == last`.

1 *Complexity:* Exactly `max((last - first) - 1, 0)` applications of the corresponding comparisons.

```
template<class ForwardIterator>
  ForwardIterator max_element(ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Compare>
  ForwardIterator max_element(ForwardIterator first, ForwardIterator last,
                              Compare comp);
```

2 *Returns:* The first iterator `i` in the range `[first,last)` such that for every iterator `j` in the range `[first,last)` the following corresponding conditions hold: `!(*i < *j)` or `comp(*i, *j) == false`. Returns `last` if `first == last`.

3 *Complexity:* Exactly `max((last - first) - 1, 0)` applications of the corresponding comparisons.

```
template<class ForwardIterator>
  pair<ForwardIterator, ForwardIterator>
    minmax_element(ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Compare>
  pair<ForwardIterator, ForwardIterator>
    minmax_element(ForwardIterator first, ForwardIterator last, Compare comp);
```

4 *Returns:* `make_pair(first, first)` if `[first,last)` is empty, otherwise `make_pair(m, M)`, where `m` is the first iterator in `[first,last)` such that no iterator in the range refers to a smaller element, and where `M` is the last iterator in `[first,last)` such that no iterator in the range refers to a larger element.

5 *Complexity:* At most $max(\lfloor \frac{3}{2}(N-1) \rfloor, 0)$ applications of the corresponding predicate, where N is `distance(first, last)`.

25.204.1 Lexicographical comparison [alg.lex.comparison]

```
template<class InputIterator1, class InputIterator2>
  bool
    lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
                            InputIterator2 first2, InputIterator2 last2);

template<class InputIterator1, class InputIterator2, class Compare>
  bool
    lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
                            InputIterator2 first2, InputIterator2 last2,
                            Compare comp);
```

1 *Returns:* `true` if the sequence of elements defined by the range `[first1,last1)` is lexicographically less than the sequence of elements defined by the range `[first2,last2)` and `false` otherwise.

2 *Complexity:* At most `2*min((last1 - first1), (last2 - first2))` applications of the corresponding comparison.

3 *Remarks:* If two sequences have the same number of elements and their corresponding elements are equivalent, then neither sequence is lexicographically less than the other. If one sequence is a prefix of the other, then the shorter sequence is lexicographically less than the longer sequence. Otherwise, the lexicographical comparison of the sequences yields the same result as the comparison of the first corresponding pair of elements that are not equivalent.

```
for ( ; first1 != last1 && first2 != last2 ; ++first1, ++first2) {
  if (*first1 < *first2) return true;
  if (*first2 < *first1) return false;
}
return first1 == last1 && first2 != last2;
```

4 *Remarks:* An empty sequence is lexicographically less than any non-empty sequence, but not less than any empty sequence.

25.204.2 Permutation generators [alg.permutation.generators]

```
template<class BidirectionalIterator>
  bool next_permutation(BidirectionalIterator first,
                        BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
  bool next_permutation(BidirectionalIterator first,
                        BidirectionalIterator last, Compare comp);
```

1 *Effects:* Takes a sequence defined by the range `[first,last)` and transforms it into the next permutation. The next permutation is found by assuming that the set of all permutations is lexicographically sorted with respect to `operator<` or `comp`. If such a permutation exists, it returns `true`. Otherwise, it transforms the sequence into the smallest permutation, that is, the ascendingly sorted one, and returns `false`.

2 *Requires:* `BidirectionalIterator` shall satisfy the requirements of `ValueSwappable` (17.143.0.1).

3 *Complexity:* At most `(last - first)/2` swaps.

```
template<class BidirectionalIterator>
  bool prev_permutation(BidirectionalIterator first,
                        BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
  bool prev_permutation(BidirectionalIterator first,
                        BidirectionalIterator last, Compare comp);
```

4 *Effects:* Takes a sequence defined by the range `[first,last)` and transforms it into the previous permutation. The previous permutation is found by assuming that the set of all permutations is lexicographically sorted with respect to `operator<` or `comp`.

5 *Returns:* `true` if such a permutation exists. Otherwise, it transforms the sequence into the largest permutation, that is, the descendingly sorted one, and returns `false`.

6 *Requires:* `BidirectionalIterator` shall satisfy the requirements of `ValueSwappable` (17.143.0.1).

7 *Complexity:* At most `(last - first)/2` swaps.

25.205 C library algorithms [alg.c.library]

1 Table 113 describes some of the contents of the header `<cstdlib>`.

Table 113 — Header `<cstdlib>` synopsis

Type	Name(s)
Type:	`size_t`
Functions:	`bsearch` `qsort`

2

The contents are the same as the Standard C library header `<stdlib.h>` with the following exceptions:

1

The function signature:

```
bsearch(const void *, const void *, size_t, size_t,
  int (*)(const void *, const void *));
```

is replaced by the two declarations:

```
extern "C" void* bsearch(const void* key, const void* base,
                         size_t nmemb, size_t size,
                         int (*compar)(const void*, const void*));
extern "C++" void* bsearch(const void* key, const void* base,
                           size_t nmemb, size_t size,
                           int (*compar)(const void*, const void*));
```

both of which have the same behavior as the original declaration.

1 The function signature:

```
qsort(void *, size_t, size_t,
  int (*)(const void *, const void *));
```

is replaced by the two declarations:

```
extern "C" void qsort(void* base, size_t nmemb, size_t size,
                      int (*compar)(const void*, const void*));
extern "C++" void qsort(void* base, size_t nmemb, size_t size,
                        int (*compar)(const void*, const void*));
```

both of which have the same behavior as the original declaration. The behavior is undefined unless the objects in the array pointed to by `base` are of trivial type.

[*Note:* Because the function argument `compar()` may throw an exception, `bsearch()` and `qsort()` are allowed to propagate the exception (17.677.0.1). — *end note*]

See also: ISO C 7.10.5.

26 Numerics library [numerics]

26.1 General [numerics.general]

1 This Clause describes components that C++ programs may use to perform seminumerical operations.

2 The following subclauses describe components for complex number types, random number generation, numeric (n-at-a-time) arrays, generalized numeric algorithms, and facilities included from the ISO C library, as summarized in Table 114.

Table 114 — Numerics library summary

Subclause		Header(s)
26.2	Requirements	
26.5	Floating-Point Environment	`<cfenv>`
26.11	Complex Numbers	`<complex>`
26.45	Random number generation	`<random>`
26.129	Numeric arrays	`<valarray>`
26.144	Generalized numeric operations	`<numeric>`

26.149	C library	`<cmath>`
		`<ctgmath>`
		`<tgmath.h>`
		`<cstdlib>`

26.2 Numeric type requirements [numeric.requirements]

1 The `complex` and `valarray` components are parameterized by the type of information they contain and manipulate. A C++ program shall instantiate these components only with a type `T` that satisfies the following requirements:[272]

(1.1) — `T` is not an abstract class (it has no pure virtual member functions);

(1.2) — `T` is not a reference type;

(1.3) — `T` is not cv-qualified;

(1.4) — If `T` is a class, it has a public default constructor;

(1.5) — If `T` is a class, it has a public copy constructor with the signature `T::T(const T&)`

(1.6) — If `T` is a class, it has a public destructor;

(1.7) — If `T` is a class, it has a public assignment operator whose signature is either `T& T::operator=(const T&)` or `T& T::operator=(T)`

(1.8) — If `T` is a class, its assignment operator, copy and default constructors, and destructor shall correspond to each other in the following sense: Initialization of raw storage using the default constructor, followed by assignment, is semantically equivalent to initialization of raw storage using the copy constructor.

272) In other words, value types. These include arithmetic types, pointers, the library class `complex`, and instantiations of `valarray` for value types.

Destruction of an object, followed by initialization of its raw storage using the copy constructor, is semantically equivalent to assignment to the original object.

[*Note:* This rule states that there shall not be any subtle differences in the semantics of initialization versus assignment. This gives an implementation considerable flexibility in how arrays are initialized.

[*Example:* An implementation is allowed to initialize a `valarray` by allocating storage using the `new` operator (which implies a call to the default constructor for each element) and then assigning each element its value. Or the implementation can allocate raw storage and use the copy constructor to initialize each element. — *end example*]

If the distinction between initialization and assignment is important for a class, or if it fails to satisfy any of the other conditions listed above, the programmer should use `vector` (23.696.1) instead of `valarray` for that class; — *end note*]

(1.9) — If `T` is a class, it does not overload unary `operator&`.

2 If any operation on `T` throws an exception the effects are undefined.

3

In addition, many member and related functions of `valarray<T>` can be successfully instantiated and will exhibit well-defined behavior if and only if `T` satisfies additional requirements specified for each such member or related function.

1

[*Example:* It is valid to instantiate `valarray<complex>`, but `operator>()` will not be successfully instantiated for `valarray<complex>` operands, since `complex` does not have any ordering operators. — *end example*]

26.5 The floating-point environment [cfenv]

26.5.1 Header `<cfenv>` synopsis [cfenv.syn]

```
namespace std {
  // types
  typedef object type  fenv_t;
  typedef integer type fexcept_t;

  // functions
  int feclearexcept(int except);
  int fegetexceptflag(fexcept_t* pflag, int except);
  int feraiseexcept(int except);
  int fesetexceptflag(const fexcept_t* pflag, int except);
  int fetestexcept(int except);

  int fegetround(void);
  int fesetround(int mode);

  int fegetenv(fenv_t* penv);
  int feholdexcept(fenv_t* penv);
  int fesetenv(const fenv_t* penv);
  int feupdateenv(const fenv_t* penv);
}
```

1 The header also defines the macros:

```
FE_ALL_EXCEPT
FE_DIVBYZERO
```

```
FE_INEXACT
FE_INVALID
FE_OVERFLOW
FE_UNDERFLOW

FE_DOWNWARD
FE_TONEAREST
FE_TOWARDZERO
FE_UPWARD

FE_DFL_ENV
```

2

The header defines all functions, types, and macros the same as Clause 7.6 of the C standard.

1

The floating-point environment has thread storage duration (3.2291.1). The initial state for a thread's floating-point environment is the state of the floating-point environment of the thread that constructs the corresponding `std::thread` object (30.25.1) at the time it constructed the object. [*Note:* That is, the child thread gets the floating-point state of the parent thread at the time of the child's creation. — *end note*]

1 A separate floating-point environment shall be maintained for each thread. Each function accesses the environment corresponding to its calling thread.

26.11 Complex numbers [complex.numbers]

1

The header `<complex>` defines a class template, and numerous functions for representing and manipulating complex numbers.

1

The effect of instantiating the template `complex` for any type other than `float`, `double`, or `long double` is unspecified. The specializations `complex<float>`, `complex<double>`, and `complex<long double>` are literal types (3.2955).

1

If the result of a function is not mathematically defined or not in the range of representable values for its type, the behavior is undefined.

1

If `z` is an lvalue expression of type *cv* `std::complex<T>` then:

— the expression `reinterpret_cast<cv T(&)[2]>(z)` shall be well-formed,

— `reinterpret_cast<cv T(&)[2]>(z)[0]` shall designate the real part of `z`, and

— `reinterpret_cast<cv T(&)[2]>(z)[1]` shall designate the imaginary part of `z`.

Moreover, if `a` is an expression of type `cv std::complex<T>*` and the expression `a[i]` is well-defined for an integer expression `i`, then:

— `reinterpret_cast<cv T*>(a)[2*i]` shall designate the real part of `a[i]`, and

— `reinterpret_cast<cv T*>(a)[2*i + 1]` shall designate the imaginary part of `a[i]`.

26.33.1 Header <complex> synopsis [complex.syn]

```
namespace std {
  template<class T> class complex;
  template<> class complex<float>;
  template<> class complex<double>;
  template<> class complex<long double>;

  // 26.34.1, operators:
  template<class T>
    complex<T> operator+(const complex<T>&, const complex<T>&);
  template<class T> complex<T> operator+(const complex<T>&, const T&);
  template<class T> complex<T> operator+(const T&, const complex<T>&);

  template<class T> complex<T> operator-(
    const complex<T>&, const complex<T>&);
  template<class T> complex<T> operator-(const complex<T>&, const T&);
  template<class T> complex<T> operator-(const T&, const complex<T>&);

  template<class T> complex<T> operator*(
    const complex<T>&, const complex<T>&);
  template<class T> complex<T> operator*(const complex<T>&, const T&);
  template<class T> complex<T> operator*(const T&, const complex<T>&);

  template<class T> complex<T> operator/(
    const complex<T>&, const complex<T>&);
  template<class T> complex<T> operator/(const complex<T>&, const T&);
  template<class T> complex<T> operator/(const T&, const complex<T>&);

  template<class T> complex<T> operator+(const complex<T>&);
  template<class T> complex<T> operator-(const complex<T>&);

  template<class T> constexpr bool operator==(
    const complex<T>&, const complex<T>&);
  template<class T> constexpr bool operator==(const complex<T>&, const T&);
  template<class T> constexpr bool operator==(const T&, const complex<T>&);

  template<class T> constexpr bool operator!=(const complex<T>&, const complex<T>&);
  template<class T> constexpr bool operator!=(const complex<T>&, const T&);
  template<class T> constexpr bool operator!=(const T&, const complex<T>&);

  template<class T, class charT, class traits>
  basic_istream<charT, traits>&
  operator>>(basic_istream<charT, traits>&, complex<T>&);

  template<class T, class charT, class traits>
  basic_ostream<charT, traits>&
  operator<<(basic_ostream<charT, traits>&, const complex<T>&);

  // 26.34.2, values:
  template<class T> constexpr T real(const complex<T>&);
  template<class T> constexpr T imag(const complex<T>&);

  template<class T> T abs(const complex<T>&);
  template<class T> T arg(const complex<T>&);
  template<class T> T norm(const complex<T>&);
```

```
  template<class T> complex<T> conj(const complex<T>&);
  template <class T> complex<T> proj(const complex<T>&);
  template<class T> complex<T> polar(const T&, const T& = 0);

  // 26.35.1, transcendentals:
  template<class T> complex<T> acos(const complex<T>&);
  template<class T> complex<T> asin(const complex<T>&);
  template<class T> complex<T> atan(const complex<T>&);

  template<class T> complex<T> acosh(const complex<T>&);
  template<class T> complex<T> asinh(const complex<T>&);
  template<class T> complex<T> atanh(const complex<T>&);

  template<class T> complex<T> cos  (const complex<T>&);
  template<class T> complex<T> cosh (const complex<T>&);
  template<class T> complex<T> exp  (const complex<T>&);
  template<class T> complex<T> log  (const complex<T>&);
  template<class T> complex<T> log10(const complex<T>&);

  template<class T> complex<T> pow(const complex<T>&, const T&);
  template<class T> complex<T> pow(const complex<T>&, const complex<T>&);
  template<class T> complex<T> pow(const T&, const complex<T>&);

  template<class T> complex<T> sin  (const complex<T>&);
  template<class T> complex<T> sinh (const complex<T>&);
  template<class T> complex<T> sqrt (const complex<T>&);
  template<class T> complex<T> tan  (const complex<T>&);
  template<class T> complex<T> tanh (const complex<T>&);

  // 26.43.1, complex literals:
  inline namespace literals {
    inline namespace complex_literals {
      constexpr complex<long double> operator""il(long double);
      constexpr complex<long double> operator""il(unsigned long long);
      constexpr complex<double> operator""i(long double);
      constexpr complex<double> operator""i(unsigned long long);
      constexpr complex<float> operator""if(long double);
      constexpr complex<float> operator""if(unsigned long long);
    }
  }
}
```

26.33.2 Class template `complex` [complex]

```
namespace std {
  template<class T>
  class complex {
  public:
    typedef T value_type;

    constexpr complex(const T& re = T(), const T& im = T());
    constexpr complex(const complex&);
    template<class X> constexpr complex(const complex<X>&);

    constexpr T real() const;
```

```
    void real(T);
    constexpr T imag() const;
    void imag(T);

    complex<T>& operator= (const T&);
    complex<T>& operator+=(const T&);
    complex<T>& operator-=(const T&);
    complex<T>& operator*=(const T&);
    complex<T>& operator/=(const T&);

    complex& operator=(const complex&);
    template<class X> complex<T>& operator= (const complex<X>&);
    template<class X> complex<T>& operator+=(const complex<X>&);
    template<class X> complex<T>& operator-=(const complex<X>&);
    template<class X> complex<T>& operator*=(const complex<X>&);
    template<class X> complex<T>& operator/=(const complex<X>&);
  };
}
```

1 The class `complex` describes an object that can store the Cartesian components, `real()` and `imag()`, of a complex number.

26.33.3 `complex` specializations [complex.special]

```
namespace std {
  template<> class complex<float> {
  public:
    typedef float value_type;

    constexpr complex(float re = 0.0f, float im = 0.0f);
    constexpr explicit complex(const complex<double>&);
    constexpr explicit complex(const complex<long double>&);

    constexpr float real() const;
    void real(float);
    constexpr float imag() const;
    void imag(float);

    complex<float>& operator= (float);
    complex<float>& operator+=(float);
    complex<float>& operator-=(float);
    complex<float>& operator*=(float);
    complex<float>& operator/=(float);

    complex<float>& operator=(const complex<float>&);
    template<class X> complex<float>& operator= (const complex<X>&);
    template<class X> complex<float>& operator+=(const complex<X>&);
    template<class X> complex<float>& operator-=(const complex<X>&);
    template<class X> complex<float>& operator*=(const complex<X>&);
    template<class X> complex<float>& operator/=(const complex<X>&);
  };

  template<> class complex<double> {
  public:
    typedef double value_type;
```

```
    constexpr complex(double re = 0.0, double im = 0.0);
    constexpr complex(const complex<float>&);
    constexpr explicit complex(const complex<long double>&);

    constexpr double real() const;
    void real(double);
    constexpr double imag() const;
    void imag(double);

    complex<double>& operator= (double);
    complex<double>& operator+=(double);
    complex<double>& operator-=(double);
    complex<double>& operator*=(double);
    complex<double>& operator/=(double);

    complex<double>& operator=(const complex<double>&);
    template<class X> complex<double>& operator= (const complex<X>&);
    template<class X> complex<double>& operator+=(const complex<X>&);
    template<class X> complex<double>& operator-=(const complex<X>&);
    template<class X> complex<double>& operator*=(const complex<X>&);
    template<class X> complex<double>& operator/=(const complex<X>&);
  };

  template<> class complex<long double> {
  public:
    typedef long double value_type;

    constexpr complex(long double re = 0.0L, long double im = 0.0L);
    constexpr complex(const complex<float>&);
    constexpr complex(const complex<double>&);

    constexpr long double real() const;
    void real(long double);
    constexpr long double imag() const;
    void imag(long double);

    complex<long double>& operator=(const complex<long double>&);
    complex<long double>& operator= (long double);
    complex<long double>& operator+=(long double);
    complex<long double>& operator-=(long double);
    complex<long double>& operator*=(long double);
    complex<long double>& operator/=(long double);

    template<class X> complex<long double>& operator= (const complex<X>&);
    template<class X> complex<long double>& operator+=(const complex<X>&);
    template<class X> complex<long double>& operator-=(const complex<X>&);
    template<class X> complex<long double>& operator*=(const complex<X>&);
    template<class X> complex<long double>& operator/=(const complex<X>&);
  };
}
```

26.33.4 `complex` member functions [complex.members]

`template<class T> constexpr complex(const T& re = T(), const T& im = T());`

1 *Effects:* Constructs an object of class `complex`.

2 *Postcondition:* `real() == re && imag() == im`.

`constexpr T real() const;`

Returns: The value of the real component.

`void real(T val);`

Effects: Assigns `val` to the real component.

`constexpr T imag() const;`

Returns: The value of the imaginary component.

`void imag(T val);`

Effects: Assigns `val` to the imaginary component.

26.33.5 `complex` member operators [complex.member.ops]

`complex<T>& operator+=(const T& rhs);`

1 *Effects:* Adds the scalar value `rhs` to the real part of the complex value `*this` and stores the result in the real part of `*this`, leaving the imaginary part unchanged.

2 *Returns:* `*this`.

`complex<T>& operator-=(const T& rhs);`

3 *Effects:* Subtracts the scalar value `rhs` from the real part of the complex value `*this` and stores the result in the real part of `*this`, leaving the imaginary part unchanged.

4 *Returns:* `*this`.

`complex<T>& operator*=(const T& rhs);`

5 *Effects:* Multiplies the scalar value `rhs` by the complex value `*this` and stores the result in `*this`.

6 *Returns:* `*this`.

`complex<T>& operator/=(const T& rhs);`

7 *Effects:* Divides the scalar value `rhs` into the complex value `*this` and stores the result in `*this`.

8 *Returns:* `*this`.

`template<class X> complex<T>& operator+=(const complex<X>& rhs);`

9 *Effects:* Adds the complex value `rhs` to the complex value `*this` and stores the sum in `*this`.

10 *Returns:* `*this`.

`template<class X> complex<T>& operator-=(const complex<X>& rhs);`

11 *Effects:* Subtracts the complex value `rhs` from the complex value `*this` and stores the difference in `*this`.

12 *Returns:* `*this`.

`template<class X> complex<T>& operator*=(const complex<X>& rhs);`

13 *Effects:* Multiplies the complex value `rhs` by the complex value `*this` and stores the product in `*this`.

Returns: `*this`.

```
template<class X> complex<T>& operator/=(const complex<X>& rhs);
```

14

Effects: Divides the complex value `rhs` into the complex value `*this` and stores the quotient in `*this`.

1 *Returns:* `*this`.

26.34.1 `complex` non-member operations [complex.ops]

```
template<class T> complex<T> operator+(const complex<T>& lhs);
```

1 *Remarks:* unary operator.

2 *Returns:* `complex<T>(lhs)`.

```
  template<class T>
    complex<T> operator+(const complex<T>& lhs, const complex<T>& rhs);
  template<class T> complex<T> operator+(const complex<T>& lhs, const T& rhs);
  template<class T> complex<T> operator+(const T& lhs, const complex<T>& rhs);
```

3 *Returns:* `complex<T>(lhs) += rhs`.

```
template<class T> complex<T> operator-(const complex<T>& lhs);
```

4 *Remarks:* unary operator.

5 *Returns:* `complex<T>(-lhs.real(),-lhs.imag())`.

```
template<class T>
  complex<T> operator-(const complex<T>& lhs, const complex<T>& rhs);
template<class T> complex<T> operator-(const complex<T>& lhs, const T& rhs);
template<class T> complex<T> operator-(const T& lhs, const complex<T>& rhs);
```

6 *Returns:* `complex<T>(lhs) -= rhs`.

```
template<class T>
  complex<T> operator*(const complex<T>& lhs, const complex<T>& rhs);
template<class T> complex<T> operator*(const complex<T>& lhs, const T& rhs);
template<class T> complex<T> operator*(const T& lhs, const complex<T>& rhs);
```

7 *Returns:* `complex<T>(lhs) *= rhs`.

```
template<class T>
  complex<T> operator/(const complex<T>& lhs, const complex<T>& rhs);
template<class T> complex<T> operator/(const complex<T>& lhs, const T& rhs);
template<class T> complex<T> operator/(const T& lhs, const complex<T>& rhs);
```

8 *Returns:* `complex<T>(lhs) /= rhs`.

```
template<class T>
  constexpr bool operator==(const complex<T>& lhs, const complex<T>& rhs);
template<class T> constexpr bool operator==(const complex<T>& lhs, const T& rhs);
template<class T> constexpr bool operator==(const T& lhs, const complex<T>& rhs);
```

9 *Returns:* `lhs.real() == rhs.real() && lhs.imag() == rhs.imag()`.

10 *Remarks:* The imaginary part is assumed to be `T()`, or 0.0, for the `T` arguments.

```
template<class T>
  constexpr bool operator!=(const complex<T>& lhs, const complex<T>& rhs);
template<class T> constexpr bool operator!=(const complex<T>& lhs, const T& rhs);
template<class T> constexpr bool operator!=(const T& lhs, const complex<T>& rhs);
```

11 *Returns:* `rhs.real() != lhs.real() || rhs.imag() != lhs.imag()`.

```
template<class T, class charT, class traits>
basic_istream<charT, traits>&
operator>>(basic_istream<charT, traits>& is, complex<T>& x);
```

12 *Effects:* Extracts a complex number `x` of the form: `u`, `(u)`, or `(u,v)`, where `u` is the real part and `v` is the imaginary part (27.128.0.1).

13 *Requires:* The input values shall be convertible to `T`.

If bad input is encountered, calls `is.setstate(ios_base::failbit)` (which may throw `ios::failure` (27.84.0.1)).

14 *Returns:* `is`.

15 *Remarks:* This extraction is performed as a series of simpler extractions. Therefore, the skipping of whitespace is specified to be the same for each of the simpler extractions.

```
template<class T, class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& o, const complex<T>& x);
```

16 *Effects:* inserts the complex number `x` onto the stream `o` as if it were implemented as follows:

```
template<class T, class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& o, const complex<T>& x) {
  basic_ostringstream<charT, traits> s;
  s.flags(o.flags());
  s.imbue(o.getloc());
  s.precision(o.precision());
  s << '(' << x.real() << "," << x.imag() << ')';
  return o << s.str();
}
```

17 *Note:* In a locale in which comma is used as a decimal point character, the use of comma as a field separator can be ambiguous. Inserting `std::showpoint` into the output stream forces all outputs to show an explicit decimal point character; as a result, all inserted sequences of complex numbers can be extracted unambiguously.

26.34.2 `complex` value operations [complex.value.ops]

```
template<class T> constexpr T real(const complex<T>& x);
```

1 *Returns:* `x.real()`.

```
template<class T> constexpr T imag(const complex<T>& x);
```

2 *Returns:* `x.imag()`.

```
template<class T> T abs(const complex<T>& x);
```

3 *Returns:* The magnitude of `x`.

```
template<class T> T arg(const complex<T>& x);
```

4 *Returns:* The phase angle of `x`, or `atan2(imag(x), real(x))`.

```
template<class T> T norm(const complex<T>& x);
```

1 *Returns:* The squared magnitude of `x`.

```
template<class T> complex<T> conj(const complex<T>& x);
```

2 *Returns:* The complex conjugate of `x`.

```
template<class T> complex<T> proj(const complex<T>& x);
```

3 *Returns:* The projection of `x` onto the Riemann sphere.

4 *Remarks:* Behaves the same as the C function `cproj`, defined in 7.3.9.4.

```
template<class T> complex<T> polar(const T& rho, const T& theta = 0);
```

5 *Returns:* The `complex` value corresponding to a complex number whose magnitude is `rho` and whose phase angle is `theta`.

26.35.1 `complex` transcendentals [complex.transcendentals]

```
template<class T> complex<T> acos(const complex<T>& x);
```

1 *Returns:* The complex arc cosine of `x`.

2 *Remarks:* Behaves the same as C function `cacos`, defined in 7.3.5.1.

```
template<class T> complex<T> asin(const complex<T>& x);
```

3 *Returns:* The complex arc sine of `x`.

4 *Remarks:* Behaves the same as C function `casin`, defined in 7.3.5.2.

```
template<class T> complex<T> atan(const complex<T>& x);
```

5 *Returns:* The complex arc tangent of `x`.

6 *Remarks:* Behaves the same as C function `catan`, defined in 7.3.5.3.

```
template<class T> complex<T> acosh(const complex<T>& x);
```

7 *Returns:* The complex arc hyperbolic cosine of `x`.

8 *Remarks:* Behaves the same as C function `cacosh`, defined in 7.3.6.1.

```
template<class T> complex<T> asinh(const complex<T>& x);
```

9 *Returns:* The complex arc hyperbolic sine of `x`.

10 *Remarks:* Behaves the same as C function `casinh`, defined in 7.3.6.2.

```
template<class T> complex<T> atanh(const complex<T>& x);
```

11 *Returns:* The complex arc hyperbolic tangent of `x`.

12 *Remarks:* Behaves the same as C function `catanh`, defined in 7.3.6.3.

```
template<class T> complex<T> cos(const complex<T>& x);
```

13 *Returns:* The complex cosine of `x`.

```
template<class T> complex<T> cosh(const complex<T>& x);
```

14 *Returns:* The complex hyperbolic cosine of `x`.

```
template<class T> complex<T> exp(const complex<T>& x);
```

15 *Returns:* The complex base e exponential of `x`.

```
template<class T> complex<T> log(const complex<T>& x);
```

16 *Remarks:* the branch cuts are along the negative real axis.

17 *Returns:* The complex natural (base e) logarithm of `x`, in the range of a strip mathematically unbounded along the real axis and in the interval `[-i times pi,i times pi]` along the imaginary axis. When `x` is a negative real number, `imag(log(x))` is pi.

```
template<class T> complex<T> log10(const complex<T>& x);
```

18 *Remarks:* the branch cuts are along the negative real axis.

19 *Returns:* The complex common (base 10) logarithm of `x`, defined as `log(x)/log(10)`.

```
template<class T>
  complex<T> pow(const complex<T>& x, const complex<T>& y);
template<class T> complex<T> pow  (const complex<T>& x, const T& y);
template<class T> complex<T> pow  (const T& x, const complex<T>& y);
```

20 *Remarks:* the branch cuts are along the negative real axis.

21 *Returns:* The complex power of base `x` raised to the `y`-th power, defined as `exp(y*log(x))`. The value returned for `pow(0,0)` is implementation-defined.

```
template<class T> complex<T> sin  (const complex<T>& x);
```

22 *Returns:* The complex sine of `x`.

```
template<class T> complex<T> sinh (const complex<T>& x);
```

23 *Returns:* The complex hyperbolic sine of `x`.

```
template<class T> complex<T> sqrt (const complex<T>& x);
```

24 *Remarks:* the branch cuts are along the negative real axis.

25

Returns: The complex square root of `x`, in the range of the right half-plane. If the argument is a negative real number, the value returned lies on the positive imaginary axis.

```
template<class T> complex<T> tan  (const complex<T>& x);
```

1 *Returns:* The complex tangent of `x`.

```
template<class T> complex<T> tanh (const complex<T>& x);
```

2 *Returns:* The complex hyperbolic tangent of `x`.

26.36.1 Additional overloads [cmplx.over]

1

The following function templates shall have additional overloads:

```
arg                 norm
conj                proj
imag                real
```

1

The additional overloads shall be sufficient to ensure:

1. If the argument has type `long double`, then it is effectively cast to `complex<long double>`.
2. Otherwise, if the argument has type `double` or an integer type, then it is effectively cast to `complex<double>`.
3. Otherwise, if the argument has type `float`, then it is effectively cast to `complex<float>`.

1

Function template `pow` shall have additional overloads sufficient to ensure, for a call with at least one argument of type `complex<T>`:

1. If either argument has type `complex<long double>` or type `long double`, then both arguments are effectively cast to `complex<long double>`.
2. Otherwise, if either argument has type `complex<double>`, `double`, or an integer type, then both arguments are effectively cast to `complex<double>`.
3. Otherwise, if either argument has type `complex<float>` or `float`, then both arguments are effectively cast to `complex<float>`.

26.43.1 Suffixes for complex number literals [complex.literals]

1 This section describes literal suffixes for constructing complex number literals. The suffixes `i`, `il`, and `if` create complex numbers of the types `complex<double>`, `complex<long double>`, and `complex<float>` respectively, with their imaginary part denoted by the given literal number and the real part being zero.

```
constexpr complex<long double> operator""il(long double d);
constexpr complex<long double> operator""il(unsigned long long d);
```

2 *Returns:* `complex<long double>{0.0L, static_cast<long double>(d)}`.

```
constexpr complex<double> operator""i(long double d);
constexpr complex<double> operator""i(unsigned long long d);
```

3 *Returns:* `complex<double>{0.0, static_cast<double>(d)}`.

```
constexpr complex<float> operator""if(long double d);
constexpr complex<float> operator""if(unsigned long long d);
```

4 *Returns:* `complex<float>{0.0f, static_cast<float>(d)}`.

26.43.2 Header `<ccomplex>` [ccmplx]

1

The header behaves as if it simply includes the header `<complex>`.

26.45 Random number generation [rand]

1 This subclause defines a facility for generating (pseudo-)random numbers.

2

In addition to a few utilities, four categories of entities are described: *uniform random number generators*, *random number engines*, *random number engine adaptors*, and *random number distributions*. These categorizations are applicable to types that satisfy the corresponding requirements, to objects instantiated from such types, and to templates producing such types when instantiated. [*Note:* These entities are specified in

such a way as to permit the binding of any uniform random number generator object `e` as the argument to any random number distribution object `d`, thus producing a zero-argument function object such as given by `bind(d,e)`. — *end note*]

1 Each of the entities specified via this subclause has an associated arithmetic type (3.3151.1) identified as `result_type`. With `T` as the `result_type` thus associated with such an entity, that entity is characterized:

a) as *boolean* or equivalently as *boolean-valued*, if `T` is `bool`;

b) otherwise as *integral* or equivalently as *integer-valued*, if `numeric_limits<T>::is_integer` is `true`;

c) otherwise as *floating* or equivalently as *real-valued*.

If integer-valued, an entity may optionally be further characterized as *signed* or *unsigned*, according to `numeric_limits<T>::is_signed`.

2 Unless otherwise specified, all descriptions of calculations in this subclause use mathematical real numbers.

3 Throughout this subclause, the operators bitand, bitor, and xor denote the respective conventional bitwise operations. Further:

a) the operator rshift denotes a bitwise right shift with zero-valued bits appearing in the high bits of the result, and

b) the operator lshift_w denotes a bitwise left shift with zero-valued bits appearing in the low bits of the result, and whose result is always taken modulo 2^w.

26.46.1 Requirements [rand.req]

26.46.1.1 General requirements [rand.req.genl]

1

Throughout this subclause 26.45, the effect of instantiating a template:

a) that has a template type parameter named `Sseq` is undefined unless the corresponding template argument is cv-unqualified and satisfies the requirements of seed sequence (26.70.0.1).

b) that has a template type parameter named `URNG` is undefined unless the corresponding template argument is cv-unqualified and satisfies the requirements of uniform random number generator (26.75.0.1).

c) that has a template type parameter named `Engine` is undefined unless the corresponding template argument is cv-unqualified and satisfies the requirements of random number engine (26.81.0.1).

d) that has a template type parameter named `RealType` is undefined unless the corresponding template argument is cv-unqualified and is one of `float`, `double`, or `long double`.

e) that has a template type parameter named `IntType` is undefined unless the corresponding template argument is cv-unqualified and is one of `short`, `int`, `long`, `long long`, `unsigned short`, `unsigned int`, `unsigned long`, or `unsigned long long`.

f) that has a template type parameter named `UIntType` is undefined unless the corresponding template argument is cv-unqualified and is one of `unsigned short`, `unsigned int`, `unsigned long`, or `unsigned long long`.

1 Throughout this subclause 26.45, phrases of the form "`x` is an iterator of a specific kind" shall be interpreted as equivalent to the more formal requirement that "`x` is a value of a type satisfying the requirements of the specified iterator type."

2 Throughout this subclause 26.45, any constructor that can be called with a single argument and that satisfies a requirement specified in this subclause shall be declared `explicit`.

26.70.0.1 Seed sequence requirements [rand.req.seedseq]

1

A *seed sequence* is an object that consumes a sequence of integer-valued data and produces a requested number of unsigned integer values i, $0 \leq i < 2^{32}$, based on the consumed data. [*Note:* Such an object provides a mechanism to avoid replication of streams of random variates. This can be useful, for example, in applications requiring large numbers of random number engines. — *end note*]

1

A class `S` satisfies the requirements of a seed sequence if the expressions shown in Table 115 are valid and have the indicated semantics, and if `S` also satisfies all other requirements of this section 26.70.0.1. In that Table and throughout this section:

a) `T` is the type named by `S`'s associated `result_type`;

b) `q` is a value of `S` and `r` is a possibly const value of `S`;

c) `ib` and `ie` are input iterators with an unsigned integer `value_type` of at least 32 bits;

d) `rb` and `re` are mutable random access iterators with an unsigned integer `value_type` of at least 32 bits;

e) `ob` is an output iterator; and

f) `il` is a value of `initializer_list<T>`.

Table 115 — Seed sequence requirements

Expression	Return type	Pre/post-condition	Complexity
`S::result_type`	`T`	`T` is an unsigned integer type (3.3151.1) of at least 32 bits.	compile-time
`S()`		Creates a seed sequence with the same initial state as all other default-constructed seed sequences of type `S`.	constant
`S(ib,ie)`		Creates a seed sequence having internal state that depends on some or all of the bits of the supplied sequence $[\texttt{ib}, \texttt{ie})$.	$\mathcal{O}(\texttt{ie} - \texttt{ib})$
`S(il)`		Same as `S(il.begin(), il.end())`.	same as `S(il.begin(), il.end())`
`q.generate(rb,re)`	`void`	Does nothing if `rb == re`. Otherwise, fills the supplied sequence $[\texttt{rb}, \texttt{re})$ with 32-bit quantities that depend on the sequence supplied to the constructor and possibly also depend on the history of `generate`'s previous invocations.	$\mathcal{O}(\texttt{re} - \texttt{rb})$

Table 115 — Seed sequence requirements (continued)

Expression	Return type	Pre/post-condition	Complexity
`r.size()`	`size_t`	The number of 32-bit units that would be copied by a call to `r.param`.	constant
`r.param(ob)`	`void`	Copies to the given destination a sequence of 32-bit units that can be provided to the constructor of a second object of type `S`, and that would reproduce in that second object a state indistinguishable from the state of the first object.	$\mathcal{O}$(`r.size()`)

26.75.0.1 Uniform random number generator requirements [rand.req.urng]

1

A *uniform random number generator* `g` of type `G` is a function object returning unsigned integer values such that each value in the range of possible results has (ideally) equal probability of being returned. [*Note:* The degree to which `g`'s results approximate the ideal is often determined statistically. — *end note*]

1

A class `G` satisfies the requirements of a *uniform random number generator* if the expressions shown in Table 116 are valid and have the indicated semantics, and if `G` also satisfies all other requirements of this section 26.75.0.1. In that Table and throughout this section:

a) `T` is the type named by `G`'s associated `result_type`, and

b) `g` is a value of `G`.

Table 116 — Uniform random number generator requirements

Expression	Return type	Pre/post-condition	Complexity
`G::result_type`	`T`	`T` is an unsigned integer type (3.3151.1).	compile-time
`g()`	`T`	Returns a value in the closed interval [`G::min()`, `G::max()`].	amortized constant
`G::min()`	`T`	Denotes the least value potentially returned by `operator()`.	compile-time
`G::max()`	`T`	Denotes the greatest value potentially returned by `operator()`.	compile-time

1

The following relation shall hold: `G::min() < G::max()`.

26.81.0.1 Random number engine requirements [rand.req.eng]

1 A *random number engine* (commonly shortened to *engine*) `e` of type `E` is a uniform random number generator that additionally meets the requirements (*e.g.*, for seeding and for input/output) specified in this section.

2 At any given time, `e` has a state $\mathtt{e}_i$ for some integer $i \geq 0$. Upon construction, `e` has an initial state $\mathtt{e}_0$. An engine's state may be established via a constructor, a `seed` function, assignment, or a suitable `operator>>`.

3 `E`'s specification shall define:

a) the size of `E`'s state in multiples of the size of `result_type`, given as an integral constant expression;

b) the *transition algorithm* TA by which `e`'s state $\mathtt{e}_i$ is advanced to its *successor state* $\mathtt{e}_{i+1}$; and

c) the *generation algorithm* GA by which an engine's state is mapped to a value of type `result_type`.

4

A class `E` that satisfies the requirements of a uniform random number generator (26.75.0.1) also satisfies the requirements of a *random number engine* if the expressions shown in Table 117 are valid and have the indicated semantics, and if `E` also satisfies all other requirements of this section 26.81.0.1. In that Table and throughout this section:

a) `T` is the type named by `E`'s associated `result_type`;

b) `e` is a value of `E`, `v` is an lvalue of `E`, `x` and `y` are (possibly `const`) values of `E`;

c) `s` is a value of `T`;

d) `q` is an lvalue satisfying the requirements of a seed sequence (26.70.0.1);

e) `z` is a value of type `unsigned long long`;

f) `os` is an lvalue of the type of some class template specialization `basic_ostream<charT, traits>`; and

g) `is` is an lvalue of the type of some class template specialization `basic_istream<charT, traits>`;

where `charT` and `traits` are constrained according to Clause 21 and Clause 27.

Table 117 — Random number engine requirements

Expression	Return type	Pre/post-condition	Complexity
`E()`		Creates an engine with the same initial state as all other default-constructed engines of type `E`.	$\mathcal{O}$(size of state)
`E(x)`		Creates an engine that compares equal to `x`.	$\mathcal{O}$(size of state)
`E(s)`		Creates an engine with initial state determined by `s`.	$\mathcal{O}$(size of state)

Expression	Return type	Pre/post-condition	Complexity
`E(q)`[273]		Creates an engine with an initial state that depends on a sequence produced by one call to `q.generate`.	same as complexity of `q.generate` called on a sequence whose length is size of state
`e.seed()`	`void`	post: `e == E()`.	same as `E()`
`e.seed(s)`	`void`	post: `e == E(s)`.	same as `E(s)`
`e.seed(q)`	`void`	post: `e == E(q)`.	same as `E(q)`
`e()`	`T`	Advances `e`'s state $\mathtt{e}_i$ to $\mathtt{e}_{i+1}$ = $\mathsf{TA}(\mathtt{e}_i)$ and returns $\mathsf{GA}(\mathtt{e}_i)$.	per Table 116
`e.discard(z)` [274]	`void`	Advances `e`'s state $\mathtt{e}_i$ to $\mathtt{e}_{i+\mathtt{z}}$ by any means equivalent to `z` consecutive calls `e()`.	no worse than the complexity of `z` consecutive calls `e()`
`x == y`	`bool`	This operator is an equivalence relation. With S_x and S_y as the infinite sequences of values that would be generated by repeated future calls to `x()` and `y()`, respectively, returns `true` if $S_x = S_y$; else returns `false`.	$\mathcal{O}$(size of state)
`x != y`	`bool`	`!(x == y)`.	$\mathcal{O}$(size of state)
`os << x`	reference to the type of `os`	With `os.`*fmtflags* set to `ios_base::dec|ios_base::left` and the fill character set to the space character, writes to `os` the textual representation of `x`'s current state. In the output, adjacent numbers are separated by one or more space characters. post: The `os.`*fmtflags* and fill character are unchanged.	$\mathcal{O}$(size of state)

273) This constructor (as well as the subsequent corresponding `seed()` function) may be particularly useful to applications requiring a large number of independent random sequences.

274) This operation is common in user code, and can often be implemented in an engine-specific manner so as to provide significant performance improvements over an equivalent naive loop that makes `z` consecutive calls `e()`.

Expression	Return type	Pre/post-condition	Complexity
`is >> v`	reference to the type of `is`	With `is.fmtflags` set to `ios_base::dec`, sets `v`'s state as determined by reading its textual representation from `is`. If bad input is encountered, ensures that `v`'s state is unchanged by the operation and calls `is.setstate(ios::failbit)` (which may throw `ios::failure` [27.84.0.1]). If a textual representation written via `os << x` was subsequently read via `is >> v`, then `x == v` provided that there have been no intervening invocations of `x` or of `v`. pre: `is` provides a textual representation that was previously written using an output stream whose imbued locale was the same as that of `is`, and whose type's template specialization arguments `charT` and `traits` were respectively the same as those of `is`. post: The `is.`*fmtflags* are unchanged.	$\mathcal{O}$(size of state)

1 `E` shall meet the requirements of `CopyConstructible` (Table 21) and `CopyAssignable` (Table 23) types. These operations shall each be of complexity no worse than $\mathcal{O}$(size of state).

26.88.0.1 Random number engine adaptor requirements [rand.req.adapt]

1 A *random number engine adaptor* (commonly shortened to *adaptor*) `a` of type `A` is a random number engine that takes values produced by some other random number engine, and applies an algorithm to those values in order to deliver a sequence of values with different randomness properties. An engine `b` of type `B` adapted in this way is termed a *base engine* in this context. The expression `a.base()` shall be valid and shall return a const reference to `a`'s base engine.

2 The requirements of a random number engine type shall be interpreted as follows with respect to a random number engine adaptor type.

```
A::A();
```

3 *Effects:* The base engine is initialized as if by its default constructor.

```
bool operator==(const A& a1, const A& a2);
```

4 *Returns:* `true` if `a1`'s base engine is equal to `a2`'s base engine. Otherwise returns `false`.

```
A::A(result_type s);
```

5 *Effects:* The base engine is initialized with `s`.

```
template<class Sseq> void A::A(Sseq& q);
```

6 *Effects:* The base engine is initialized with `q`.

```
void seed();
```

7 *Effects:* With `b` as the base engine, invokes `b.seed()`.

```
void seed(result_type s);
```

8 *Effects:* With `b` as the base engine, invokes `b.seed(s)`.

```
template<class Sseq> void seed(Sseq& q);
```

9 *Effects:* With `b` as the base engine, invokes `b.seed(q)`.

10 `A` shall also satisfy the following additional requirements:

a) The complexity of each function shall not exceed the complexity of the corresponding function applied to the base engine.

b) The state of `A` shall include the state of its base engine. The size of `A`'s state shall be no less than the size of the base engine.

c) Copying `A`'s state (*e.g.*, during copy construction or copy assignment) shall include copying the state of the base engine of `A`.

d) The textual representation of `A` shall include the textual representation of its base engine.

26.88.0.2 Random number distribution requirements [rand.req.dist]

1 A *random number distribution* (commonly shortened to *distribution*) `d` of type `D` is a function object returning values that are distributed according to an associated mathematical *probability density function* $p(z)$ or according to an associated *discrete probability function* $P(z_i)$. A distribution's specification identifies its associated probability function $p(z)$ or $P(z_i)$.

2 An associated probability function is typically expressed using certain externally-supplied quantities known as the *parameters of the distribution*. Such distribution parameters are identified in this context by writing, for example, $p(z \mid a, b)$ or $P(z_i \mid a, b)$, to name specific parameters, or by writing, for example, $p(z \mid \{\mathtt{p}\})$ or $P(z_i \mid \{\mathtt{p}\})$, to denote a distribution's parameters `p` taken as a whole.

3

A class `D` satisfies the requirements of a *random number distribution* if the expressions shown in Table 118 are valid and have the indicated semantics, and if `D` and its associated types also satisfy all other requirements of this section 26.88.0.2. In that Table and throughout this section,

a) `T` is the type named by `D`'s associated `result_type`;

b) `P` is the type named by `D`'s associated `param_type`;

c) `d` is a value of `D`, and `x` and `y` are (possibly `const`) values of `D`;

d) `glb` and `lub` are values of `T` respectively corresponding to the greatest lower bound and the least upper bound on the values potentially returned by `d`'s `operator()`, as determined by the current values of `d`'s parameters;

e) `p` is a (possibly `const`) value of `P`;

f) `g`, `g1`, and `g2` are lvalues of a type satisfying the requirements of a uniform random number generator [26.75.0.1];

g) `os` is an lvalue of the type of some class template specialization `basic_ostream<charT, traits>`; and

h) `is` is an lvalue of the type of some class template specialization `basic_istream<charT, traits>`;

where `charT` and `traits` are constrained according to Clauses 21 and 27.

Table 118 — Random number distribution requirements

Expression	Return type	Pre/post-condition	Complexity
`D::result_type`	`T`	`T` is an arithmetic type (3.3151.1).	compile-time
`D::param_type`	`P`		compile-time
`D()`		Creates a distribution whose behavior is indistinguishable from that of any other newly default-constructed distribution of type `D`.	constant
`D(p)`		Creates a distribution whose behavior is indistinguishable from that of a distribution newly constructed directly from the values used to construct `p`.	same as `p`'s construction
`d.reset()`	`void`	Subsequent uses of `d` do not depend on values produced by any engine prior to invoking `reset`.	constant
`x.param()`	`P`	Returns a value `p` such that `D(p).param() == p`.	no worse than the complexity of `D(p)`
`d.param(p)`	`void`	post: `d.param() == p`.	no worse than the complexity of `D(p)`
`d(g)`	`T`	With `p` = `d.param()`, the sequence of numbers returned by successive invocations with the same object `g` is randomly distributed according to the associated $p(z \mid \{\mathtt{p}\})$ or $P(z_i \mid \{\mathtt{p}\})$ function.	amortized constant number of invocations of `g`
`d(g,p)`	`T`	The sequence of numbers returned by successive invocations with the same objects `g` and `p` is randomly distributed according to the associated $p(z \mid \{\mathtt{p}\})$ or $P(z_i \mid \{\mathtt{p}\})$ function.	amortized constant number of invocations of `g`
`x.min()`	`T`	Returns `glb`.	constant
`x.max()`	`T`	Returns `lub`.	constant

Expression	Return type	Pre/post-condition	Complexity
`x == y`	`bool`	This operator is an equivalence relation. Returns `true` if `x.param() == y.param()` and $S_1 = S_2$, where S_1 and S_2 are the infinite sequences of values that would be generated, respectively, by repeated future calls to `x(g1)` and `y(g2)` whenever `g1 == g2`. Otherwise returns `false`.	constant
`x != y`	`bool`	`!(x == y)`.	same as `x == y`.
`os << x`	reference to the type of `os`	Writes to `os` a textual representation for the parameters and the additional internal data of `x`. post: The `os.`*fmtflags* and fill character are unchanged.	
`is >> d`	reference to the type of `is`	Restores from `is` the parameters and additional internal data of the lvalue `d`. If bad input is encountered, ensures that `d` is unchanged by the operation and calls `is.setstate(ios::failbit)` (which may throw `ios::failure` [27.84.0.1]). pre: `is` provides a textual representation that was previously written using an `os` whose imbued locale and whose type's template specialization arguments `charT` and `traits` were the same as those of `is`. post: The `is.`*fmtflags* are unchanged.	

1 `D` shall satisfy the requirements of `CopyConstructible` (Table 21) and `CopyAssignable` (Table 23) types.

2 The sequence of numbers produced by repeated invocations of `d(g)` shall be independent of any invocation of `os << d` or of any `const` member function of `D` between any of the invocations `d(g)`.

3 If a textual representation is written using `os << x` and that representation is restored into the same or a different object `y` of the same type using `is >> y`, repeated invocations of `y(g)` shall produce the same sequence of numbers as would repeated invocations of `x(g)`.

4

It is unspecified whether `D::param_type` is declared as a (nested) `class` or via a `typedef`. In this subclause 26.45, declarations of `D::param_type` are in the form of `typedef`s for convenience of exposition only.

1 `P` shall satisfy the requirements of `CopyConstructible` (Table 21), `CopyAssignable` (Table 23), and `EqualityComparable`

(Table 17) types.

2

For each of the constructors of `D` taking arguments corresponding to parameters of the distribution, `P` shall have a corresponding constructor subject to the same requirements and taking arguments identical in number, type, and default values. Moreover, for each of the member functions of `D` that return values corresponding to parameters of the distribution, `P` shall have a corresponding member function with the identical name, type, and semantics.

1 `P` shall have a declaration of the form

```
typedef  D  distribution_type;
```

26.97.1 Header <random> synopsis [rand.synopsis]

```
#include <initializer_list>

namespace std {

 // 26.98.0.1, class template linear_congruential_engine
 template<class UIntType, UIntType a, UIntType c, UIntType m>
   class linear_congruential_engine;

 // 26.100.0.1, class template mersenne_twister_engine
 template<class UIntType, size_t w, size_t n, size_t m, size_t r,
          UIntType a, size_t u, UIntType d, size_t s,
          UIntType b, size_t t,
          UIntType c, size_t l, UIntType f>
   class mersenne_twister_engine;

 // 26.101.0.1, class template subtract_with_carry_engine
 template<class UIntType, size_t w, size_t s, size_t r>
   class subtract_with_carry_engine;

 // 26.101.1.2, class template discard_block_engine
 template<class Engine, size_t p, size_t r>
   class discard_block_engine;

 // 26.101.1.3, class template independent_bits_engine
 template<class Engine, size_t w, class UIntType>
   class independent_bits_engine;

 // 26.101.1.4, class template shuffle_order_engine
 template<class Engine, size_t k>
   class shuffle_order_engine;

 // 26.101.2, engines and engine adaptors with predefined parameters
 typedef see below minstd_rand0;
 typedef see below minstd_rand;
 typedef see below mt19937;
 typedef see below mt19937_64;
 typedef see below ranlux24_base;
 typedef see below ranlux48_base;
 typedef see below ranlux24;
 typedef see below ranlux48;
 typedef see below knuth_b;
```

```
typedef see below default_random_engine;

// 26.105.1, class random_device
class random_device;

// 26.111.1.1, class seed_seq
class seed_seq;

// 26.111.1.2, function template generate_canonical
template<class RealType, size_t bits, class URNG>
  RealType generate_canonical(URNG& g);

// 26.118.0.1.1, class template uniform_int_distribution
template<class IntType = int>
  class uniform_int_distribution;

// 26.120.0.0.1, class template uniform_real_distribution
template<class RealType = double>
  class uniform_real_distribution;

// 26.122.0.1.1, class bernoulli_distribution
class bernoulli_distribution;

// 26.122.0.1.2, class template binomial_distribution
template<class IntType = int>
  class binomial_distribution;

// 26.122.0.1.3, class template geometric_distribution
template<class IntType = int>
  class geometric_distribution;

// 26.122.0.1.4, class template negative_binomial_distribution
template<class IntType = int>
  class negative_binomial_distribution;

// 26.122.0.2.1, class template poisson_distribution
template<class IntType = int>
  class poisson_distribution;

// 26.122.0.2.2, class template exponential_distribution
template<class RealType = double>
  class exponential_distribution;

// 26.122.0.2.3, class template gamma_distribution
template<class RealType = double>
  class gamma_distribution;

// 26.122.0.2.4, class template weibull_distribution
template<class RealType = double>
  class weibull_distribution;

// 26.122.0.2.5, class template extreme_value_distribution
template<class RealType = double>
  class extreme_value_distribution;
```

```
// 26.122.0.3.1, class template normal_distribution
template<class RealType = double>
  class normal_distribution;

// 26.128.0.0.1, class template lognormal_distribution
template<class RealType = double>
  class lognormal_distribution;

// 26.128.0.0.2, class template chi_squared_distribution
template<class RealType = double>
  class chi_squared_distribution;

// 26.128.0.0.3, class template cauchy_distribution
template<class RealType = double>
  class cauchy_distribution;

// 26.128.0.0.4, class template fisher_f_distribution
template<class RealType = double>
  class fisher_f_distribution;

// 26.128.0.0.5, class template student_t_distribution
template<class RealType = double>
  class student_t_distribution;

// 26.128.0.1.1, class template discrete_distribution
template<class IntType = int>
  class discrete_distribution;

// 26.128.0.1.2, class template piecewise_constant_distribution
template<class RealType = double>
  class piecewise_constant_distribution;

// 26.128.0.1.3, class template piecewise_linear_distribution
template<class RealType = double>
  class piecewise_linear_distribution;

} // namespace std
```

26.97.2 Random number engine class templates [rand.eng]

1 Each type instantiated from a class template specified in this section 26.97.2 satisfies the requirements of a random number engine (26.81.0.1) type.

2 Except where specified otherwise, the complexity of each function specified in this section 26.97.2 is constant.

3 Except where specified otherwise, no function described in this section 26.97.2 throws an exception.

4 Descriptions are provided in this section 26.97.2 only for engine operations that are not described in 26.81.0.1 or for operations where there is additional semantic information. In particular, declarations for copy constructors, for copy assignment operators, for streaming operators, and for equality and inequality operators are not shown in the synopses.

5 Each template specified in this section 26.97.2 requires one or more relationships, involving the value(s) of its non-type template parameter(s), to hold. A program instantiating any of these templates is ill-formed if any such required relationship fails to hold.

6

For every random number engine and for every random number engine adaptor `X` defined in this subclause (26.97.2) and in sub-clause 26.101.1:

— if the constructor

```
template <class Sseq> explicit X(Sseq& q);
```

is called with a type `Sseq` that does not qualify as a seed sequence, then this constructor shall not participate in overload resolution;

— if the member function

```
template <class Sseq> void seed(Sseq& q);
```

is called with a type `Sseq` that does not qualify as a seed sequence, then this function shall not participate in overload resolution.

The extent to which an implementation determines that a type cannot be a seed sequence is unspecified, except that as a minimum a type shall not qualify as a seed sequence if it is implicitly convertible to `X::result_type`.

26.98.0.1 Class template `linear_congruential_engine` [rand.eng.lcong]

1 A `linear_congruential_engine` random number engine produces unsigned integer random numbers. The state $\mathtt{x}_i$ of a `linear_congruential_engine` object `x` is of size 1 and consists of a single integer. The transition algorithm is a modular linear function of the form $\mathsf{TA}(\mathtt{x}_i) = (a \cdot \mathtt{x}_i + c) \bmod m$; the generation algorithm is $\mathsf{GA}(\mathtt{x}_i) = \mathtt{x}_{i+1}$.

```
template<class UIntType, UIntType a, UIntType c, UIntType m>
 class linear_congruential_engine
{
public:
 // types
 typedef UIntType result_type;

 // engine characteristics
 static constexpr result_type multiplier = a;
 static constexpr result_type increment = c;
 static constexpr result_type modulus = m;
 static constexpr result_type min() { return c == 0u ? 1u: 0u; }
 static constexpr result_type max() { return m - 1u; }
 static constexpr result_type default_seed = 1u;

 // constructors and seeding functions
 explicit linear_congruential_engine(result_type s = default_seed);
 template<class Sseq> explicit linear_congruential_engine(Sseq& q);
 void seed(result_type s = default_seed);
 template<class Sseq> void seed(Sseq& q);

 // generating functions
 result_type operator()();
 void discard(unsigned long long z);
};
```

2 If the template parameter `m` is 0, the modulus m used throughout this section 26.98.0.1 is `numeric_limits<result_type>::max()` plus 1. [*Note:* m need not be representable as a value of type `result_type`. — *end note*]

3 If the template parameter `m` is not 0, the following relations shall hold: `a < m` and `c < m`.

4 The textual representation consists of the value of $\mathtt{x}_i$.

```
explicit linear_congruential_engine(result_type s = default_seed);
```

5 *Effects:* Constructs a `linear_congruential_engine` object. If $c \bmod m$ is 0 and `s` mod m is 0, sets the engine's state to 1, otherwise sets the engine's state to `s` mod m.

```
template<class Sseq> explicit linear_congruential_engine(Sseq& q);
```

1 *Effects:* Constructs a `linear_congruential_engine` object. With $k = \left\lceil \frac{\log_2 m}{32} \right\rceil$ and a an array (or equivalent) of length $k+3$, invokes `q.generate(`$a+0$`, `$a+k+3$`)` and then computes $S = \left(\sum_{j=0}^{k-1} a_{j+3} \cdot 2^{32j}\right) \bmod m$. If $c \bmod m$ is 0 and S is 0, sets the engine's state to 1, else sets the engine's state to S.

26.100.0.1 Class template `mersenne_twister_engine` [rand.eng.mers]

1

A `mersenne_twister_engine` random number engine[275] produces unsigned integer random numbers in the closed interval $[0, 2^w - 1]$. The state $\mathtt{x}_i$ of a `mersenne_twister_engine` object `x` is of size n and consists of a sequence X of n values of the type delivered by `x`; all subscripts applied to X are to be taken modulo n.

1 The transition algorithm employs a twisted generalized feedback shift register defined by shift values n and m, a twist value r, and a conditional xor-mask a. To improve the uniformity of the result, the bits of the raw shift register are additionally *tempered* (*i.e.*, scrambled) according to a bit-scrambling matrix defined by values u, d, s, b, t, c, and ℓ.

The state transition is performed as follows:

a) Concatenate the upper $w-r$ bits of X_{i-n} with the lower r bits of X_{i+1-n} to obtain an unsigned integer value Y.

b) With $\alpha = a \cdot (Y \,\mathsf{bitand}\, 1)$, set X_i to $X_{i+m-n} \,\mathsf{xor}\, (Y \,\mathsf{rshift}\, 1) \,\mathsf{xor}\, \alpha$.

The sequence X is initialized with the help of an initialization multiplier f.

2 The generation algorithm determines the unsigned integer values z_1, z_2, z_3, z_4 as follows, then delivers z_4 as its result:

a) Let $z_1 = X_i \,\mathsf{xor}\, \big((X_i \,\mathsf{rshift}\, u) \,\mathsf{bitand}\, d\big)$.

b) Let $z_2 = z_1 \,\mathsf{xor}\, \big((z_1 \,\mathsf{lshift}_w\, s) \,\mathsf{bitand}\, b\big)$.

c) Let $z_3 = z_2 \,\mathsf{xor}\, \big((z_2 \,\mathsf{lshift}_w\, t) \,\mathsf{bitand}\, c\big)$.

d) Let $z_4 = z_3 \,\mathsf{xor}\, (z_3 \,\mathsf{rshift}\, \ell)$.

```
template<class UIntType, size_t w, size_t n, size_t m, size_t r,
         UIntType a, size_t u, UIntType d, size_t s,
         UIntType b, size_t t,
         UIntType c, size_t l, UIntType f>
 class mersenne_twister_engine
{
```

275) The name of this engine refers, in part, to a property of its period: For properly-selected values of the parameters, the period is closely related to a large Mersenne prime number.

```
public:
 // types
 typedef UIntType result_type;

 // engine characteristics
 static constexpr size_t word_size = w;
 static constexpr size_t state_size = n;
 static constexpr size_t shift_size = m;
 static constexpr size_t mask_bits = r;
 static constexpr UIntType xor_mask = a;
 static constexpr size_t tempering_u = u;
 static constexpr UIntType tempering_d = d;
 static constexpr size_t tempering_s = s;
 static constexpr UIntType tempering_b = b;
 static constexpr size_t tempering_t = t;
 static constexpr UIntType tempering_c = c;
 static constexpr size_t tempering_l = l;
 static constexpr UIntType initialization_multiplier = f;
 static constexpr result_type min() { return 0; }
 static constexpr result_type max() { return  2^w - 1; }
 static constexpr result_type default_seed = 5489u;

 // constructors and seeding functions
 explicit mersenne_twister_engine(result_type value = default_seed);
 template<class Sseq> explicit mersenne_twister_engine(Sseq& q);
 void seed(result_type value = default_seed);
 template<class Sseq> void seed(Sseq& q);

 // generating functions
 result_type operator()();
 void discard(unsigned long long z);
};
```

3 The following relations shall hold: `0 < m`, `m <= n`, `2u < w`, `r <= w`, `u <= w`, `s <= w`, `t <= w`, `l <= w`, `w <= numeric_limits<UIntType>::digits`, `a <= (1u<<w) - 1u`, `b <= (1u<<w) - 1u`, `c <= (1u<<w) - 1u`, `d <= (1u<<w) - 1u`, and `f <= (1u<<w) - 1u`.

4 The textual representation of $\mathtt{x}_i$ consists of the values of $X_{i-n}, \ldots, X_{i-1}$, in that order.

`explicit mersenne_twister_engine(result_type value = default_seed);`

5 *Effects:* Constructs a `mersenne_twister_engine` object. Sets X_{-n} to `value` mod 2^w. Then, iteratively for $i = 1-n, \ldots, -1$, sets X_i to

$$\left[f \cdot \left(X_{i-1} \,\mathsf{xor}\, \left(X_{i-1} \,\mathsf{rshift}\, (w-2)\right)\right) + i \bmod n\right] \bmod 2^w \ .$$

6 *Complexity:* $\mathscr{O}(n)$.

`template<class Sseq> explicit mersenne_twister_engine(Sseq& q);`

7 *Effects:* Constructs a `mersenne_twister_engine` object. With $k = \lceil w/32 \rceil$ and a an array (or equivalent) of length $n \cdot k$, invokes `q.generate(`$a + 0$`, `$a + n \cdot k$`)` and then, iteratively for $i = -n, \ldots, -1$, sets X_i to $\left(\sum_{j=0}^{k-1} a_{k(i+n)+j} \cdot 2^{32j}\right) \bmod 2^w$. Finally, if the most significant $w - r$ bits of X_{-n} are zero, and if each of the other resulting X_i is 0, changes X_{-n} to 2^{w-1}.

26.101.0.1 Class template `subtract_with_carry_engine` [rand.eng.sub]

1 A `subtract_with_carry_engine` random number engine produces unsigned integer random numbers.

2 The state $\mathtt{x}_i$ of a `subtract_with_carry_engine` object `x` is of size $\mathscr{O}(r)$, and consists of a sequence X of r integer values $0 \leq X_i < m = 2^w$; all subscripts applied to X are to be taken modulo r. The state $\mathtt{x}_i$ additionally consists of an integer c (known as the *carry*) whose value is either 0 or 1.

3 The state transition is performed as follows:

a) Let $Y = X_{i-s} - X_{i-r} - c$.

b) Set X_i to $y = Y \bmod m$. Set c to 1 if $Y < 0$, otherwise set c to 0.

[*Note:* This algorithm corresponds to a modular linear function of the form $\mathsf{TA}(\mathtt{x}_i) = (a \cdot \mathtt{x}_i) \bmod b$, where b is of the form $m^r - m^s + 1$ and $a = b - (b-1)/m$. — *end note*]

4 The generation algorithm is given by $\mathsf{GA}(\mathtt{x}_i) = y$, where y is the value produced as a result of advancing the engine's state as described above.

```
template<class UIntType, size_t w, size_t s, size_t r>
 class subtract_with_carry_engine
{
public:
 // types
 typedef UIntType result_type;

 // engine characteristics
 static constexpr size_t word_size = w;
 static constexpr size_t short_lag = s;
 static constexpr size_t long_lag = r;
 static constexpr result_type min() { return 0; }
 static constexpr result_type max() { return m − 1; }
 static constexpr result_type default_seed = 19780503u;

 // constructors and seeding functions
 explicit subtract_with_carry_engine(result_type value = default_seed);
 template<class Sseq> explicit subtract_with_carry_engine(Sseq& q);
 void seed(result_type value = default_seed);
 template<class Sseq> void seed(Sseq& q);

 // generating functions
 result_type operator()();
 void discard(unsigned long long z);
};
```

5 The following relations shall hold: `0u < s`, `s < r`, `0 < w`, and `w <= numeric_limits<UIntType>::digits`.

6 The textual representation consists of the values of $X_{i-r}, \ldots, X_{i-1}$, in that order, followed by c.

`explicit subtract_with_carry_engine(result_type value = default_seed);`

7 *Effects:* Constructs a `subtract_with_carry_engine` object. Sets the values of $X_{-r}, \ldots, X_{-1}$, in that order, as specified below. If X_{-1} is then 0, sets c to 1; otherwise sets c to 0.

To set the values X_k, first construct `e`, a `linear_congruential_engine` object, as if by the following definition:

```
linear_congruential_engine<result_type,
                           40014u,0u,2147483563u> e(value == 0u ? default_seed : value);
```

Then, to set each X_k, obtain new values $z_0, \ldots, z_{n-1}$ from $n = \lceil w/32 \rceil$ successive invocations of `e` taken modulo 2^{32}. Set X_k to $\left(\sum_{j=0}^{n-1} z_j \cdot 2^{32j}\right) \bmod m$.

8 *Complexity:* Exactly $n \cdot$ `r` invocations of `e`.

```
template<class Sseq> explicit subtract_with_carry_engine(Sseq& q);
```

9 *Effects:* Constructs a `subtract_with_carry_engine` object. With $k = \lceil w/32 \rceil$ and a an array (or equivalent) of length $r \cdot k$, invokes `q.generate(`$a+0$`, `$a+r \cdot k$`)` and then, iteratively for $i = -r, \ldots, -1$, sets X_i to $\left(\sum_{j=0}^{k-1} a_{k(i+r)+j} \cdot 2^{32j}\right) \bmod m$. If X_{-1} is then 0, sets c to 1; otherwise sets c to 0.

26.101.1 Random number engine adaptor class templates [rand.adapt]

26.101.1.1 In general [rand.adapt.general]

1 Each type instantiated from a class template specified in this section 26.97.2 satisfies the requirements of a random number engine adaptor (26.88.0.1) type.

2 Except where specified otherwise, the complexity of each function specified in this section 26.101.1 is constant.

3 Except where specified otherwise, no function described in this section 26.101.1 throws an exception.

4 Descriptions are provided in this section 26.101.1 only for adaptor operations that are not described in section 26.88.0.1 or for operations where there is additional semantic information. In particular, declarations for copy constructors, for copy assignment operators, for streaming operators, and for equality and inequality operators are not shown in the synopses.

5 Each template specified in this section 26.101.1 requires one or more relationships, involving the value(s) of its non-type template parameter(s), to hold. A program instantiating any of these templates is ill-formed if any such required relationship fails to hold.

26.101.1.2 Class template `discard_block_engine` [rand.adapt.disc]

1 A `discard_block_engine` random number engine adaptor produces random numbers selected from those produced by some base engine e. The state $\mathtt{x}_i$ of a `discard_block_engine` engine adaptor object `x` consists of the state $\mathtt{e}_i$ of its base engine `e` and an additional integer n. The size of the state is the size of e's state plus 1.

2 The transition algorithm discards all but $r > 0$ values from each block of $p \geq r$ values delivered by e. The state transition is performed as follows: If $n \geq r$, advance the state of `e` from $\mathtt{e}_i$ to $\mathtt{e}_{i+p-r}$ and set n to 0. In any case, then increment n and advance `e`'s then-current state $\mathtt{e}_j$ to $\mathtt{e}_{j+1}$.

3 The generation algorithm yields the value returned by the last invocation of `e()` while advancing `e`'s state as described above.

```
template<class Engine, size_t p, size_t r>
 class discard_block_engine
{
public:
 // types
 typedef typename Engine::result_type result_type;

 // engine characteristics
 static constexpr size_t block_size = p;
 static constexpr size_t used_block = r;
 static constexpr result_type min() { return Engine::min(); }
 static constexpr result_type max() { return Engine::max(); }

 // constructors and seeding functions
 discard_block_engine();
```

```
  explicit discard_block_engine(const Engine& e);
  explicit discard_block_engine(Engine&& e);
  explicit discard_block_engine(result_type s);
  template<class Sseq> explicit discard_block_engine(Sseq& q);
  void seed();
  void seed(result_type s);
  template<class Sseq> void seed(Sseq& q);

  // generating functions
  result_type operator()();
  void discard(unsigned long long z);

  // property functions
  const Engine& base() const noexcept { return e; };

 private:
  Engine e;    // exposition only
  int n;       // exposition only
 };
```

4 The following relations shall hold: `0 < r` and `r <= p`.

5 The textual representation consists of the textual representation of `e` followed by the value of `n`.

6 In addition to its behavior pursuant to section 26.88.0.1, each constructor that is not a copy constructor sets `n` to 0.

26.101.1.3 Class template `independent_bits_engine` [rand.adapt.ibits]

1 An `independent_bits_engine` random number engine adaptor combines random numbers that are produced by some base engine e, so as to produce random numbers with a specified number of bits w. The state $\mathtt{x}_i$ of an `independent_bits_engine` engine adaptor object `x` consists of the state $\mathtt{e}_i$ of its base engine `e`; the size of the state is the size of e's state.

2 The transition and generation algorithms are described in terms of the following integral constants:

a) Let $R =$ `e.max() - e.min() + 1` and $m = \lfloor \log_2 R \rfloor$.

b) With n as determined below, let $w_0 = \lfloor w/n \rfloor$, $n_0 = n - w \bmod n$, $y_0 = 2^{w_0} \lfloor R/2^{w_0} \rfloor$, and $y_1 = 2^{w_0+1} \lfloor R/2^{w_0+1} \rfloor$.

c) Let $n = \lceil w/m \rceil$ if and only if the relation $R - y_0 \leq \lfloor y_0/n \rfloor$ holds as a result. Otherwise let $n = 1 + \lceil w/m \rceil$.

[*Note:* The relation $w = n_0 w_0 + (n - n_0)(w_0 + 1)$ always holds. — *end note*]

3 The transition algorithm is carried out by invoking `e()` as often as needed to obtain n_0 values less than y_0 +`e.min()` and $n - n_0$ values less than $y_1 +$ `e.min()`.

4 The generation algorithm uses the values produced while advancing the state as described above to yield a quantity S obtained as if by the following algorithm:

```
S = 0;
for (k = 0; k ≠ n0; k += 1)  {
 do u = e() - e.min(); while (u ≥ y0);
 S = 2^w0 · S + u mod 2^w0;
}
for (k = n0; k ≠ n; k += 1)  {
```

```
 do u = e() - e.min(); while (u ≥ y1);
 S = 2^(w0+1) · S + u mod 2^(w0+1);
}

template<class Engine, size_t w, class UIntType>
class independent_bits_engine
{
public:
 // types
 typedef UIntType result_type;

 // engine characteristics
 static constexpr result_type min() { return 0; }
 static constexpr result_type max() { return 2^w − 1; }

 // constructors and seeding functions
 independent_bits_engine();
 explicit independent_bits_engine(const Engine& e);
 explicit independent_bits_engine(Engine&& e);
 explicit independent_bits_engine(result_type s);
 template<class Sseq> explicit independent_bits_engine(Sseq& q);
 void seed();
 void seed(result_type s);
 template<class Sseq> void seed(Sseq& q);

 // generating functions
 result_type operator()();
 void discard(unsigned long long z);

 // property functions
 const Engine& base() const noexcept { return e; };

private:
 Engine e;    // exposition only
};
```

5 The following relations shall hold: `0 < w` and `w <= numeric_limits<result_type>::digits`.

6 The textual representation consists of the textual representation of `e`.

26.101.1.4 Class template `shuffle_order_engine` [rand.adapt.shuf]

1 A `shuffle_order_engine` random number engine adaptor produces the same random numbers that are produced by some base engine e, but delivers them in a different sequence. The state $\mathtt{x}_i$ of a `shuffle_order_engine` engine adaptor object `x` consists of the state $\mathtt{e}_i$ of its base engine `e`, an additional value Y of the type delivered by `e`, and an additional sequence V of k values also of the type delivered by `e`. The size of the state is the size of e's state plus $k + 1$.

2 The transition algorithm permutes the values produced by e. The state transition is performed as follows:

a) Calculate an integer $j = \left\lfloor \frac{k \cdot (Y - e_{\min})}{e_{\max} - e_{\min} + 1} \right\rfloor$.

b) Set Y to V_j and then set V_j to `e()`.

3 The generation algorithm yields the last value of `Y` produced while advancing `e`'s state as described above.

```
template<class Engine, size_t k>
 class shuffle_order_engine
{
public:
 // types
 typedef typename Engine::result_type result_type;

 // engine characteristics
 static constexpr size_t table_size = k;
 static constexpr result_type min() { return Engine::min(); }
 static constexpr result_type max() { return Engine::max(); }

 // constructors and seeding functions
 shuffle_order_engine();
 explicit shuffle_order_engine(const Engine& e);
 explicit shuffle_order_engine(Engine&& e);
 explicit shuffle_order_engine(result_type s);
 template<class Sseq> explicit shuffle_order_engine(Sseq& q);
 void seed();
 void seed(result_type s);
 template<class Sseq> void seed(Sseq& q);

 // generating functions
 result_type operator()();
 void discard(unsigned long long z);

 // property functions
 const Engine& base() const noexcept { return e; };

private:
 Engine e;           // exposition only
 result_type Y;      // exposition only
 result_type V[k];   // exposition only
};
```

4 The following relation shall hold: `0 < k`.

5 The textual representation consists of the textual representation of `e`, followed by the `k` values of V, followed by the value of Y.

6 In addition to its behavior pursuant to section 26.88.0.1, each constructor that is not a copy constructor initializes `V[0]`, . . . , `V[k-1]` and Y, in that order, with values returned by successive invocations of `e()`.

26.101.2 Engines and engine adaptors with predefined parameters [rand.predef]

```
typedef linear_congruential_engine<uint_fast32_t, 16807, 0, 2147483647>
      minstd_rand0;
```

1 *Required behavior:* The 10000 th consecutive invocation of a default-constructed object of type `minstd_rand0` shall produce the value 1043618065.

```
typedef linear_congruential_engine<uint_fast32_t, 48271, 0, 2147483647>
      minstd_rand;
```

2 *Required behavior:* The 10000 th consecutive invocation of a default-constructed object of type `minstd_rand` shall produce the value 399268537.

```
typedef mersenne_twister_engine<uint_fast32_t,
        32,624,397,31,0x9908b0df,11,0xffffffff,7,0x9d2c5680,15,0xefc60000,18,1812433253>
        mt19937;
```

3 *Required behavior:* The 10000 th consecutive invocation of a default-constructed object of type `mt19937` shall produce the value 4123659995.

```
typedef mersenne_twister_engine<uint_fast64_t,
        64,312,156,31,0xb5026f5aa96619e9,29,
        0x5555555555555555,17,
        0x71d67fffeda60000,37,
        0xfff7eee000000000,43,
        6364136223846793005>
        mt19937_64;
```

1 *Required behavior:* The 10000 th consecutive invocation of a default-constructed object of type `mt19937_64` shall produce the value 9981545732273789042.

```
typedef subtract_with_carry_engine<uint_fast32_t, 24, 10, 24>
        ranlux24_base;
```

2 *Required behavior:* The 10000 th consecutive invocation of a default-constructed object of type `ranlux24_base` shall produce the value 7937952.

```
typedef subtract_with_carry_engine<uint_fast64_t, 48, 5, 12>
        ranlux48_base;
```

3 *Required behavior:* The 10000 th consecutive invocation of a default-constructed object of type `ranlux48_base` shall produce the value 61839128582725.

```
typedef discard_block_engine<ranlux24_base, 223, 23>
        ranlux24;
```

4 *Required behavior:* The 10000 th consecutive invocation of a default-constructed object of type `ranlux24` shall produce the value 9901578.

```
typedef discard_block_engine<ranlux48_base, 389, 11>
        ranlux48;
```

5 *Required behavior:* The 10000 th consecutive invocation of a default-constructed object of type `ranlux48` shall produce the value 249142670248501.

```
typedef shuffle_order_engine<minstd_rand0,256>
        knuth_b;
```

6 *Required behavior:* The 10000 th consecutive invocation of a default-constructed object of type `knuth_b` shall produce the value 1112339016.

```
typedef implementation-defined
        default_random_engine;
```

7 *Remark:* The choice of engine type named by this `typedef` is implementation-defined. [*Note:* The implementation may select this type on the basis of performance, size, quality, or any combination of such factors, so as to provide at least acceptable engine behavior for relatively casual, inexpert, and/or lightweight use. Because different implementations may select different underlying engine types, code that uses this `typedef` need not generate identical sequences across implementations. — *end note*]

26.105.1 Class random_device [rand.device]

1

A random_device uniform random number generator produces non-deterministic random numbers.

1

If implementation limitations prevent generating non-deterministic random numbers, the implementation may employ a random number engine.

```
class random_device
{
public:
 // types
 typedef unsigned int result_type;

 // generator characteristics
 static constexpr result_type min() { return numeric_limits<result_type>::min(); }
 static constexpr result_type max() { return numeric_limits<result_type>::max(); }

 // constructors
 explicit random_device(const string& token = implementation-defined);

 // generating functions
 result_type operator()();

 // property functions
 double entropy() const noexcept;

 // no copy functions
 random_device(const random_device& ) = delete;
 void operator=(const random_device& ) = delete;
};
```

`explicit random_device(const string& token = implementation-defined);`

1 *Effects:* Constructs a random_device non-deterministic uniform random number generator object. The semantics and default value of the token parameter are implementation-defined.[276]

2

Throws: A value of an implementation-defined type derived from exception if the random_device could not be initialized.

`double entropy() const noexcept;`

1 *Returns:* If the implementation employs a random number engine, returns 0.0. Otherwise, returns an entropy estimate[277] for the random numbers returned by operator(), in the range min() to $\log_2(\texttt{max()} + 1)$.

`result_type operator()();`

2 *Returns:* A non-deterministic random value, uniformly distributed between min() and max(), inclusive. It is implementation-defined how these values are generated.

1 *Throws:* A value of an implementation-defined type derived from exception if a random number could not be obtained.

276) The parameter is intended to allow an implementation to differentiate between different sources of randomness.

277) If a device has n states whose respective probabilities are $P_0, \ldots, P_{n-1}$, the device entropy S is defined as $S = -\sum_{i=0}^{n-1} P_i \cdot \log P_i$.

26.111.1 Utilities [rand.util]

26.111.1.1 Class seed_seq [rand.util.seedseq]

```
class seed_seq
{
public:
 // types
 typedef uint_least32_t result_type;

 // constructors
 seed_seq();
 template<class T>
   seed_seq(initializer_list<T> il);
 template<class InputIterator>
   seed_seq(InputIterator begin, InputIterator end);

 // generating functions
 template<class RandomAccessIterator>
   void generate(RandomAccessIterator begin, RandomAccessIterator end);

 // property functions
 size_t size() const;
 template<class OutputIterator>
   void param(OutputIterator dest) const;

 // no copy functions
 seed_seq(const seed_seq& ) = delete;
 void operator=(const seed_seq& ) = delete;

private:
 vector<result_type> v;    // exposition only
};
```

`seed_seq();`

1 *Effects:* Constructs a `seed_seq` object as if by default-constructing its member `v`.

2 *Throws:* Nothing.

`template<class T>`
`seed_seq(initializer_list<T> il);`

3 *Requires:* `T` shall be an integer type.

4 *Effects:* Same as `seed_seq(il.begin(), il.end())`.

`template<class InputIterator>`
`seed_seq(InputIterator begin, InputIterator end);`

5 *Requires:* `InputIterator` shall satisfy the requirements of an input iterator (Table 107) type. Moreover, `iterator_traits<InputIterator>::value_type` shall denote an integer type.

6 *Effects:* Constructs a `seed_seq` object by the following algorithm:

```
for( InputIterator s = begin; s != end; ++s)
 v.push_back((*s) mod 2^32);
```

`template<class RandomAccessIterator>`
`void generate(RandomAccessIterator begin, RandomAccessIterator end);`

7 *Requires:* `RandomAccessIterator` shall meet the requirements of a mutable random access iterator (Table 111) type. Moreover, `iterator_traits<RandomAccessIterator>::value_type` shall denote an unsigned integer type capable of accommodating 32-bit quantities.

8 *Effects:* Does nothing if `begin == end`. Otherwise, with $s =$ `v.size()` and $n =$ `end` $-$ `begin`, fills the supplied range [`begin`, `end`) according to the following algorithm in which each operation is to be carried out modulo 2^{32}, each indexing operator applied to `begin` is to be taken modulo n, and $T(x)$ is defined as x xor (x rshift 27):

a) By way of initialization, set each element of the range to the value `0x8b8b8b8b`. Additionally, for use in subsequent steps, let $p = (n-t)/2$ and let $q = p + t$, where

$$t = (n \geq 623)\ ?\ 11 : (n \geq 68)\ ?\ 7 : (n \geq 39)\ ?\ 5 : (n \geq 7)\ ?\ 3 : (n-1)/2;$$

b) With m as the larger of $s+1$ and n, transform the elements of the range: iteratively for $k = 0, \ldots, m-1$, calculate values

$$\begin{aligned} r_1 &= 1664525 \cdot \mathtt{T}\,(\mathtt{begin}[k] \text{ xor } \mathtt{begin}[k+p] \text{ xor } \mathtt{begin}[k-1]) \\ r_2 &= r_1 + \begin{cases} s & , k = 0 \\ k \bmod n + \mathtt{v}[k-1] & , 0 < k \leq s \\ k \bmod n & , s < k \end{cases} \end{aligned}$$

and, in order, increment `begin`$[k+p]$ by r_1, increment `begin`$[k+q]$ by r_2, and set `begin[k]` to r_2.

c) Transform the elements of the range again, beginning where the previous step ended: iteratively for $k = m, \ldots, m+n-1$, calculate values

$$\begin{aligned} r_3 &= 1566083941 \cdot \mathtt{T}\,(\mathtt{begin}[k] + \mathtt{begin}[k+p] + \mathtt{begin}[k-1]) \\ r_4 &= r_3 - (k \bmod n) \end{aligned}$$

and, in order, update `begin`$[k+p]$ by xoring it with r_3, update `begin`$[k+q]$ by xoring it with r_4, and set `begin`$[k]$ to r_4.

9 *Throws:* What and when `RandomAccessIterator` operations of `begin` and `end` throw.

```
size_t size() const;
```

10 *Returns:* The number of 32-bit units that would be returned by a call to `param()`.

11 *Throws:* Nothing.

12 *Complexity:* Constant time.

```
template<class OutputIterator>
  void param(OutputIterator dest) const;
```

13 *Requires:* `OutputIterator` shall satisfy the requirements of an output iterator (Table 108) type. Moreover, the expression `*dest = rt` shall be valid for a value `rt` of type `result_type`.

14 *Effects:* Copies the sequence of prepared 32-bit units to the given destination, as if by executing the following statement:

```
copy(v.begin(), v.end(), dest);
```

15 *Throws:* What and when `OutputIterator` operations of `dest` throw.

26.111.1.2 Function template `generate_canonical` [rand.util.canonical]

1

Each function instantiated from the template described in this section 26.111.1.2 maps the result of one or more invocations of a supplied uniform random number generator `g` to one member of the specified `RealType` such that, if the values g_i produced by `g` are uniformly distributed, the instantiation's results t_j, $0 \le t_j < 1$, are distributed as uniformly as possible as specified below.

1 [*Note:* Obtaining a value in this way can be a useful step in the process of transforming a value generated by a uniform random number generator into a value that can be delivered by a random number distribution. — *end note*]

```
template<class RealType, size_t bits, class URNG>
 RealType generate_canonical(URNG& g);
```

2 *Complexity:* Exactly $k = \max(1, \lceil b/\log_2 R \rceil)$ invocations of `g`, where b[278] is the lesser of `numeric_limits<RealType>::digits` and `bits`, and R is the value of `g.max()` − `g.min()` + 1.

3 *Effects:* Invokes `g()` k times to obtain values $g_0, \ldots, g_{k-1}$, respectively. Calculates a quantity

$$S = \sum_{i=0}^{k-1} (g_i - \texttt{g.min()}) \cdot R^i$$

using arithmetic of type `RealType`.

4 *Returns:* S/R^k.

5 *Throws:* What and when `g` throws.

26.113.1 Random number distribution class templates [rand.dist]

26.113.1.1 In general [rand.dist.general]

1 Each type instantiated from a class template specified in this section 26.113.1 satisfies the requirements of a random number distribution (26.88.0.2) type.

2 Descriptions are provided in this section 26.113.1 only for distribution operations that are not described in 26.88.0.2 or for operations where there is additional semantic information. In particular, declarations for copy constructors, for copy assignment operators, for streaming operators, and for equality and inequality operators are not shown in the synopses.

3

The algorithms for producing each of the specified distributions are implementation-defined.

1 The value of each probability density function $p(z)$ and of each discrete probability function $P(z_i)$ specified in this section is 0 everywhere outside its stated domain.

26.118.0.1 Uniform distributions [rand.dist.uni]

26.118.0.1.1 Class template `uniform_int_distribution` [rand.dist.uni.int]

1

A `uniform_int_distribution` random number distribution produces random integers i, $a \le i \le b$, distributed according to the constant discrete probability function

$$P(i \mid a, b) = 1/(b - a + 1) \ .$$

278) b is introduced to avoid any attempt to produce more bits of randomness than can be held in `RealType`.

```
template<class IntType = int>
 class uniform_int_distribution
{
public:
 // types
 typedef IntType result_type;
 typedef unspecified param_type;

 // constructors and reset functions
 explicit uniform_int_distribution(IntType a = 0, IntType b = numeric_limits<IntType>::max());
 explicit uniform_int_distribution(const param_type& parm);
 void reset();

 // generating functions
 template<class URNG>
   result_type operator()(URNG& g);
 template<class URNG>
   result_type operator()(URNG& g, const param_type& parm);

 // property functions
 result_type a() const;
 result_type b() const;
 param_type param() const;
 void param(const param_type& parm);
 result_type min() const;
 result_type max() const;
};
```

`explicit uniform_int_distribution(IntType a = 0, IntType b = numeric_limits<IntType>::max());`

1 *Requires:* `a` $\leq$ `b`.

2 *Effects:* Constructs a `uniform_int_distribution` object; `a` and `b` correspond to the respective parameters of the distribution.

`result_type a() const;`

3 *Returns:* The value of the `a` parameter with which the object was constructed.

`result_type b() const;`

4 *Returns:* The value of the `b` parameter with which the object was constructed.

26.120.0.0.1 Class template `uniform_real_distribution` [rand.dist.uni.real]

1

A `uniform_real_distribution` random number distribution produces random numbers x, $a \leq x < b$, distributed according to the constant probability density function

$$p(x \mid a, b) = 1/(b - a) \ .$$

```
template<class RealType = double>
 class uniform_real_distribution
{
public:
 // types
 typedef RealType result_type;
```

```
  typedef unspecified param_type;

  // constructors and reset functions
  explicit uniform_real_distribution(RealType a = 0.0, RealType b = 1.0);
  explicit uniform_real_distribution(const param_type& parm);
  void reset();

  // generating functions
  template<class URNG>
    result_type operator()(URNG& g);
  template<class URNG>
    result_type operator()(URNG& g, const param_type& parm);

  // property functions
  result_type a() const;
  result_type b() const;
  param_type param() const;
  void param(const param_type& parm);
  result_type min() const;
  result_type max() const;
 };
```

`explicit uniform_real_distribution(RealType a = 0.0, RealType b = 1.0);`

1 *Requires:* `a` $\leq$ `b` and `b` $-$ `a` $\leq$ `numeric_limits<RealType>::max()`.

2 *Effects:* Constructs a `uniform_real_distribution` object; `a` and `b` correspond to the respective parameters of the distribution.

`result_type a() const;`

3 *Returns:* The value of the `a` parameter with which the object was constructed.

`result_type b() const;`

4 *Returns:* The value of the `b` parameter with which the object was constructed.

26.122.0.1 Bernoulli distributions [rand.dist.bern]

26.122.0.1.1 Class `bernoulli_distribution` [rand.dist.bern.bernoulli]

1 A `bernoulli_distribution` random number distribution produces `bool` values b distributed according to the discrete probability function

$$P(b \mid p) = \begin{cases} p & \text{if} \quad b = \texttt{true} \\ 1 - p & \text{if} \quad b = \texttt{false} \end{cases} .$$

```
 class bernoulli_distribution
 {
 public:
  // types
  typedef bool result_type;
  typedef unspecified param_type;

  // constructors and reset functions
  explicit bernoulli_distribution(double p = 0.5);
  explicit bernoulli_distribution(const param_type& parm);
```

```
  void reset();

  // generating functions
  template<class URNG>
    result_type operator()(URNG& g);
  template<class URNG>
    result_type operator()(URNG& g, const param_type& parm);

  // property functions
  double p() const;
  param_type param() const;
  void param(const param_type& parm);
  result_type min() const;
  result_type max() const;
 };
```

`explicit bernoulli_distribution(double p = 0.5);`

2 *Requires:* $0 \leq \mathtt{p} \leq 1$.

3 *Effects:* Constructs a `bernoulli_distribution` object; `p` corresponds to the parameter of the distribution.

`double p() const;`

4 *Returns:* The value of the `p` parameter with which the object was constructed.

26.122.0.1.2 Class template `binomial_distribution` [rand.dist.bern.bin]

1 A `binomial_distribution` random number distribution produces integer values $i \geq 0$ distributed according to the discrete probability function

$$P(i \mid t, p) = \binom{t}{i} \cdot p^i \cdot (1-p)^{t-i} \ .$$

```
template<class IntType = int>
 class binomial_distribution
{
public:
 // types
 typedef IntType result_type;
 typedef unspecified param_type;

 // constructors and reset functions
 explicit binomial_distribution(IntType t = 1, double p = 0.5);
 explicit binomial_distribution(const param_type& parm);
 void reset();

 // generating functions
 template<class URNG>
   result_type operator()(URNG& g);
 template<class URNG>
   result_type operator()(URNG& g, const param_type& parm);

 // property functions
 IntType t() const;
 double p() const;
```

```
  param_type param() const;
  void param(const param_type& parm);
  result_type min() const;
  result_type max() const;
 };
```

```
explicit binomial_distribution(IntType t = 1, double p = 0.5);
```

2 *Requires:* $0 \leq \mathtt{p} \leq 1$ and $0 \leq \mathtt{t}$.

3 *Effects:* Constructs a `binomial_distribution` object; `t` and `p` correspond to the respective parameters of the distribution.

```
IntType t() const;
```

4 *Returns:* The value of the `t` parameter with which the object was constructed.

```
double p() const;
```

5 *Returns:* The value of the `p` parameter with which the object was constructed.

26.122.0.1.3 Class template `geometric_distribution` [rand.dist.bern.geo]

1 A `geometric_distribution` random number distribution produces integer values $i \geq 0$ distributed according to the discrete probability function

$$P(i \mid p) = p \cdot (1 - p)^i \ .$$

```
template<class IntType = int>
 class geometric_distribution
{
public:
 // types
 typedef IntType result_type;
 typedef unspecified param_type;

 // constructors and reset functions
 explicit geometric_distribution(double p = 0.5);
 explicit geometric_distribution(const param_type& parm);
 void reset();

 // generating functions
 template<class URNG>
   result_type operator()(URNG& g);
 template<class URNG>
   result_type operator()(URNG& g, const param_type& parm);

 // property functions
 double p() const;
 param_type param() const;
 void param(const param_type& parm);
 result_type min() const;
 result_type max() const;
 };
```

```
explicit geometric_distribution(double p = 0.5);
```

2 *Requires:* $0 < \mathtt{p} < 1$.

3 *Effects:* Constructs a `geometric_distribution` object; `p` corresponds to the parameter of the distribution.

```
double p() const;
```

4 *Returns:* The value of the `p` parameter with which the object was constructed.

26.122.0.1.4 Class template `negative_binomial_distribution` [rand.dist.bern.negbin]

1 A `negative_binomial_distribution` random number distribution produces random integers $i \geq 0$ distributed according to the discrete probability function

$$P(i \mid k, p) = \binom{k+i-1}{i} \cdot p^k \cdot (1-p)^i \ .$$

```
template<class IntType = int>
 class negative_binomial_distribution
{
public:
 // types
 typedef IntType  result_type;
 typedef unspecified param_type;

 // constructor and reset functions
 explicit negative_binomial_distribution(IntType k = 1, double p = 0.5);
 explicit negative_binomial_distribution(const param_type& parm);
 void reset();

 // generating functions
 template<class URNG>
   result_type operator()(URNG& g);
 template<class URNG>
   result_type operator()(URNG& g, const param_type& parm);

 // property functions
 IntType k() const;
 double p() const;
 param_type param() const;
 void param(const param_type& parm);
 result_type min() const;
 result_type max() const;
};
```

```
explicit negative_binomial_distribution(IntType k = 1, double p = 0.5);
```

2 *Requires:* $0 < \mathtt{p} \leq 1$ and $0 < \mathtt{k}$.

3 *Effects:* Constructs a `negative_binomial_distribution` object; `k` and `p` correspond to the respective parameters of the distribution.

```
IntType k() const;
```

4 *Returns:* The value of the `k` parameter with which the object was constructed.

```
double p() const;
```

5 *Returns:* The value of the `p` parameter with which the object was constructed.

26.122.0.2 Poisson distributions [rand.dist.pois]

26.122.0.2.1 Class template `poisson_distribution` [rand.dist.pois.poisson]

1 A `poisson_distribution` random number distribution produces integer values $i \geq 0$ distributed according to the discrete probability function

$$P(i \mid \mu) = \frac{e^{-\mu}\mu^i}{i\,!} \ .$$

The distribution parameter μ is also known as this distribution's *mean* .

```
template<class IntType = int>
 class poisson_distribution
{
public:
 // types
 typedef IntType result_type;
 typedef unspecified param_type;

 // constructors and reset functions
 explicit poisson_distribution(double mean = 1.0);
 explicit poisson_distribution(const param_type& parm);
 void reset();

 // generating functions
 template<class URNG>
   result_type operator()(URNG& g);
 template<class URNG>
   result_type operator()(URNG& g, const param_type& parm);

 // property functions
 double mean() const;
 param_type param() const;
 void param(const param_type& parm);
 result_type min() const;
 result_type max() const;
};
```

`explicit poisson_distribution(double mean = 1.0);`

2 *Requires:* $0 <$ `mean`.

3 *Effects:* Constructs a `poisson_distribution` object; `mean` corresponds to the parameter of the distribution.

`double mean() const;`

4 *Returns:* The value of the `mean` parameter with which the object was constructed.

26.122.0.2.2 Class template `exponential_distribution` [rand.dist.pois.exp]

1 An `exponential_distribution` random number distribution produces random numbers $x > 0$ distributed according to the probability density function

$$p(x \mid \lambda) = \lambda e^{-\lambda x} \ .$$

```
template<class RealType = double>
 class exponential_distribution
{
```

```
public:
 // types
 typedef RealType result_type;
 typedef unspecified param_type;

 // constructors and reset functions
 explicit exponential_distribution(RealType lambda = 1.0);
 explicit exponential_distribution(const param_type& parm);
 void reset();

 // generating functions
 template<class URNG>
   result_type operator()(URNG& g);
 template<class URNG>
   result_type operator()(URNG& g, const param_type& parm);

 // property functions
 RealType lambda() const;
 param_type param() const;
 void param(const param_type& parm);
 result_type min() const;
 result_type max() const;
};
```

`explicit exponential_distribution(RealType lambda = 1.0);`

2 *Requires:* $0 <$ `lambda`.

3 *Effects:* Constructs a `exponential_distribution` object; `lambda` corresponds to the parameter of the distribution.

`RealType lambda() const;`

4 *Returns:* The value of the `lambda` parameter with which the object was constructed.

26.122.0.2.3 Class template `gamma_distribution` [rand.dist.pois.gamma]

1 A `gamma_distribution` random number distribution produces random numbers $x > 0$ distributed according to the probability density function

$$p(x \mid \alpha, \beta) = \frac{e^{-x/\beta}}{\beta^\alpha \cdot \Gamma(\alpha)} \cdot x^{\alpha-1} .$$

```
template<class RealType = double>
 class gamma_distribution
{
public:
 // types
 typedef RealType result_type;
 typedef unspecified param_type;

 // constructors and reset functions
 explicit gamma_distribution(RealType alpha = 1.0, RealType beta = 1.0);
 explicit gamma_distribution(const param_type& parm);
 void reset();
```

```
  // generating functions
  template<class URNG>
    result_type operator()(URNG& g);
  template<class URNG>
    result_type operator()(URNG& g, const param_type& parm);

  // property functions
  RealType alpha() const;
  RealType beta() const;
  param_type param() const;
  void param(const param_type& parm);
  result_type min() const;
  result_type max() const;
};
```

```
explicit gamma_distribution(RealType alpha = 1.0, RealType beta = 1.0);
```

2 *Requires:* $0 <$ `alpha` and $0 <$ `beta`.

3 *Effects:* Constructs a `gamma_distribution` object; `alpha` and `beta` correspond to the parameters of the distribution.

```
RealType alpha() const;
```

4 *Returns:* The value of the `alpha` parameter with which the object was constructed.

```
RealType beta() const;
```

5 *Returns:* The value of the `beta` parameter with which the object was constructed.

26.122.0.2.4 Class template `weibull_distribution` [rand.dist.pois.weibull]

1 A `weibull_distribution` random number distribution produces random numbers $x \geq 0$ distributed according to the probability density function

$$p(x \mid a, b) = \frac{a}{b} \cdot \left(\frac{x}{b}\right)^{a-1} \cdot \exp\left(-\left(\frac{x}{b}\right)^{a}\right) .$$

```
template<class RealType = double>
 class weibull_distribution
{
public:
 // types
 typedef RealType result_type;
 typedef unspecified param_type;

 // constructor and reset functions
 explicit weibull_distribution(RealType a = 1.0, RealType b = 1.0);
 explicit weibull_distribution(const param_type& parm);
 void reset();

 // generating functions
 template<class URNG>
   result_type operator()(URNG& g);
 template<class URNG>
   result_type operator()(URNG& g, const param_type& parm);
```

```
  // property functions
  RealType a() const;
  RealType b() const;
  param_type param() const;
  void param(const param_type& parm);
  result_type min() const;
  result_type max() const;
 };
```

```
explicit weibull_distribution(RealType a = 1.0, RealType b = 1.0);
```

2 *Requires:* $0 <$ `a` and $0 <$ `b`.

3 *Effects:* Constructs a `weibull_distribution` object; `a` and `b` correspond to the respective parameters of the distribution.

```
RealType a() const;
```

4 *Returns:* The value of the `a` parameter with which the object was constructed.

```
RealType b() const;
```

5 *Returns:* The value of the `b` parameter with which the object was constructed.

26.122.0.2.5 Class template `extreme_value_distribution` [rand.dist.pois.extreme]

1 An `extreme_value_distribution` random number distribution produces random numbers x distributed according to the probability density function[279]

$$p(x \mid a, b) = \frac{1}{b} \cdot \exp\left(\frac{a-x}{b} - \exp\left(\frac{a-x}{b}\right)\right) .$$

```
 template<class RealType = double>
  class extreme_value_distribution
 {
 public:
  // types
  typedef RealType result_type;
  typedef unspecified param_type;

  // constructor and reset functions
  explicit extreme_value_distribution(RealType a = 0.0, RealType b = 1.0);
  explicit extreme_value_distribution(const param_type& parm);
  void reset();

  // generating functions
  template<class URNG>
    result_type operator()(URNG& g);
  template<class URNG>
    result_type operator()(URNG& g, const param_type& parm);

  // property functions
  RealType a() const;
  RealType b() const;
```

279) The distribution corresponding to this probability density function is also known (with a possible change of variable) as the Gumbel Type I, the log-Weibull, or the Fisher-Tippett Type I distribution.

```
  param_type param() const;
  void param(const param_type& parm);
  result_type min() const;
  result_type max() const;
 };
```

```
explicit extreme_value_distribution(RealType a = 0.0, RealType b = 1.0);
```

2 *Requires:* $0 < \texttt{b}$.

3 *Effects:* Constructs an `extreme_value_distribution` object; `a` and `b` correspond to the respective parameters of the distribution.

```
RealType a() const;
```

4 *Returns:* The value of the `a` parameter with which the object was constructed.

```
RealType b() const;
```

5 *Returns:* The value of the `b` parameter with which the object was constructed.

26.122.0.3 Normal distributions [rand.dist.norm]

26.122.0.3.1 Class template `normal_distribution` [rand.dist.norm.normal]

1

A `normal_distribution` random number distribution produces random numbers x distributed according to the probability density function

$$p(x \mid \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \cdot \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right).$$

The distribution parameters μ and σ are also known as this distribution's *mean* and *standard deviation* .

```
template<class RealType = double>
 class normal_distribution
{
public:
 // types
 typedef RealType result_type;
 typedef unspecified param_type;

 // constructors and reset functions
 explicit normal_distribution(RealType mean = 0.0, RealType stddev = 1.0);
 explicit normal_distribution(const param_type& parm);
 void reset();

 // generating functions
 template<class URNG>
   result_type operator()(URNG& g);
 template<class URNG>
   result_type operator()(URNG& g, const param_type& parm);

 // property functions
 RealType mean() const;
 RealType stddev() const;
 param_type param() const;
 void param(const param_type& parm);
```

```
  result_type min() const;
  result_type max() const;
 };
```

```
explicit normal_distribution(RealType mean = 0.0, RealType stddev = 1.0);
```

1 *Requires:* $0 <$ `stddev`.

2 *Effects:* Constructs a `normal_distribution` object; `mean` and `stddev` correspond to the respective parameters of the distribution.

```
RealType mean() const;
```

3 *Returns:* The value of the `mean` parameter with which the object was constructed.

```
RealType stddev() const;
```

4 *Returns:* The value of the `stddev` parameter with which the object was constructed.

26.128.0.0.1 Class template `lognormal_distribution` [rand.dist.norm.lognormal]

1 A `lognormal_distribution` random number distribution produces random numbers $x > 0$ distributed according to the probability density function

$$p(x \mid m, s) = \frac{1}{sx\sqrt{2\pi}} \cdot \exp\left(-\frac{(\ln x - m)^2}{2s^2}\right).$$

```
template<class RealType = double>
 class lognormal_distribution
{
public:
 // types
 typedef RealType result_type;
 typedef unspecified param_type;

 // constructor and reset functions
 explicit lognormal_distribution(RealType m = 0.0, RealType s = 1.0);
 explicit lognormal_distribution(const param_type& parm);
 void reset();

 // generating functions
 template<class URNG>
   result_type operator()(URNG& g);
 template<class URNG>
   result_type operator()(URNG& g, const param_type& parm);

 // property functions
 RealType m() const;
 RealType s() const;
 param_type param() const;
 void param(const param_type& parm);
 result_type min() const;
 result_type max() const;
};
```

```
explicit lognormal_distribution(RealType m = 0.0, RealType s = 1.0);
```

2 *Requires:* $0 <$ `s`.

3 *Effects:* Constructs a `lognormal_distribution` object; `m` and `s` correspond to the respective parameters of the distribution.

```
RealType m() const;
```

4 *Returns:* The value of the `m` parameter with which the object was constructed.

```
RealType s() const;
```

5 *Returns:* The value of the `s` parameter with which the object was constructed.

26.128.0.0.2 Class template `chi_squared_distribution` [rand.dist.norm.chisq]

1 A `chi_squared_distribution` random number distribution produces random numbers $x > 0$ distributed according to the probability density function

$$p(x \mid n) = \frac{x^{(n/2)-1} \cdot e^{-x/2}}{\Gamma(n/2) \cdot 2^{n/2}} \ .$$

```
template<class RealType = double>
 class chi_squared_distribution
{
public:
 // types
 typedef RealType result_type;
 typedef unspecified param_type;

 // constructor and reset functions
 explicit chi_squared_distribution(RealType n = 1);
 explicit chi_squared_distribution(const param_type& parm);
 void reset();

 // generating functions
 template<class URNG>
   result_type operator()(URNG& g);
 template<class URNG>
   result_type operator()(URNG& g, const param_type& parm);

 // property functions
 RealType n() const;
 param_type param() const;
 void param(const param_type& parm);
 result_type min() const;
 result_type max() const;
};
```

```
explicit chi_squared_distribution(RealType n = 1);
```

2 *Requires:* $0 <$ `n`.

3 *Effects:* Constructs a `chi_squared_distribution` object; `n` corresponds to the parameter of the distribution.

```
RealType n() const;
```

4 *Returns:* The value of the `n` parameter with which the object was constructed.

26.128.0.0.3 Class template `cauchy_distribution` [rand.dist.norm.cauchy]

1 A `cauchy_distribution` random number distribution produces random numbers x distributed according to the probability density function

$$p(x \mid a, b) = \left(\pi b \left(1 + \left(\frac{x-a}{b}\right)^2\right)\right)^{-1} .$$

```
template<class RealType = double>
 class cauchy_distribution
{
public:
 // types
 typedef RealType result_type;
 typedef unspecified param_type;

 // constructor and reset functions
 explicit cauchy_distribution(RealType a = 0.0, RealType b = 1.0);
 explicit cauchy_distribution(const param_type& parm);
 void reset();

 // generating functions
 template<class URNG>
   result_type operator()(URNG& g);
 template<class URNG>
   result_type operator()(URNG& g, const param_type& parm);

 // property functions
 RealType a() const;
 RealType b() const;
 param_type param() const;
 void param(const param_type& parm);
 result_type min() const;
 result_type max() const;
};
```

`explicit cauchy_distribution(RealType a = 0.0, RealType b = 1.0);`

2 *Requires:* $0 <$ `b`.

3 *Effects:* Constructs a `cauchy_distribution` object; `a` and `b` correspond to the respective parameters of the distribution.

`RealType a() const;`

4 *Returns:* The value of the `a` parameter with which the object was constructed.

`RealType b() const;`

5 *Returns:* The value of the `b` parameter with which the object was constructed.

26.128.0.0.4 Class template `fisher_f_distribution` [rand.dist.norm.f]

1 A `fisher_f_distribution` random number distribution produces random numbers $x \geq 0$ distributed according to the probability density function

$$p(x \mid m, n) = \frac{\Gamma\big((m+n)/2\big)}{\Gamma(m/2)\,\Gamma(n/2)} \cdot \left(\frac{m}{n}\right)^{m/2} \cdot x^{(m/2)-1} \cdot \left(1 + \frac{mx}{n}\right)^{-(m+n)/2} .$$

```
template<class RealType = double>
 class fisher_f_distribution
{
public:
 // types
 typedef RealType result_type;
 typedef unspecified param_type;

 // constructor and reset functions
 explicit fisher_f_distribution(RealType m = 1, RealType n = 1);
 explicit fisher_f_distribution(const param_type& parm);
 void reset();

 // generating functions
 template<class URNG>
   result_type operator()(URNG& g);
 template<class URNG>
   result_type operator()(URNG& g, const param_type& parm);

 // property functions
 RealType m() const;
 RealType n() const;
 param_type param() const;
 void param(const param_type& parm);
 result_type min() const;
 result_type max() const;
};
```

`explicit fisher_f_distribution(RealType m = 1, RealType n = 1);`

2 *Requires:* $0 <$ `m` and $0 <$ `n`.

3 *Effects:* Constructs a `fisher_f_distribution` object; `m` and `n` correspond to the respective parameters of the distribution.

`RealType m() const;`

4 *Returns:* The value of the `m` parameter with which the object was constructed.

`RealType n() const;`

5 *Returns:* The value of the `n` parameter with which the object was constructed.

26.128.0.0.5 Class template `student_t_distribution` [rand.dist.norm.t]

1 A `student_t_distribution` random number distribution produces random numbers x distributed according to the probability density function

$$p(x \mid n) = \frac{1}{\sqrt{n\pi}} \cdot \frac{\Gamma\big((n+1)/2\big)}{\Gamma(n/2)} \cdot \left(1 + \frac{x^2}{n}\right)^{-(n+1)/2} .$$

```
template<class RealType = double>
 class student_t_distribution
{
public:
 // types
 typedef RealType result_type;
```

```
  typedef unspecified param_type;

  // constructor and reset functions
  explicit student_t_distribution(RealType n = 1);
  explicit student_t_distribution(const param_type& parm);
  void reset();

  // generating functions
  template<class URNG>
    result_type operator()(URNG& g);
  template<class URNG>
    result_type operator()(URNG& g, const param_type& parm);

  // property functions
  RealType n() const;
  param_type param() const;
  void param(const param_type& parm);
  result_type min() const;
  result_type max() const;
 };
```

`explicit student_t_distribution(RealType n = 1);`

2 *Requires:* $0 <$ `n`.

3 *Effects:* Constructs a `student_t_distribution` object; `n` corresponds to the parameter of the distribution.

`RealType n() const;`

4 *Returns:* The value of the `n` parameter with which the object was constructed.

26.128.0.1 Sampling distributions [rand.dist.samp]

26.128.0.1.1 Class template `discrete_distribution` [rand.dist.samp.discrete]

1 A `discrete_distribution` random number distribution produces random integers i, $0 \le i < n$, distributed according to the discrete probability function

$$P(i \,|\, p_0, \ldots, p_{n-1}) = p_i \ .$$

2 Unless specified otherwise, the distribution parameters are calculated as: $p_k = w_k/S$ for $k = 0, \ldots, n-1$, in which the values w_k, commonly known as the *weights* , shall be non-negative, non-NaN, and non-infinity. Moreover, the following relation shall hold: $0 < S = w_0 + \cdots + w_{n-1}$.

```
 template<class IntType = int>
  class discrete_distribution
 {
 public:
  // types
  typedef IntType result_type;
  typedef unspecified param_type;

  // constructor and reset functions
  discrete_distribution();
  template<class InputIterator>
    discrete_distribution(InputIterator firstW, InputIterator lastW);
```

```
  discrete_distribution(initializer_list<double> wl);
  template<class UnaryOperation>
    discrete_distribution(size_t nw, double xmin, double xmax, UnaryOperation fw);
  explicit discrete_distribution(const param_type& parm);
  void reset();

  // generating functions
  template<class URNG>
    result_type operator()(URNG& g);
  template<class URNG>
    result_type operator()(URNG& g, const param_type& parm);

  // property functions
  vector<double> probabilities() const;
  param_type param() const;
  void param(const param_type& parm);
  result_type min() const;
  result_type max() const;
 };
```

`discrete_distribution();`

3 *Effects:* Constructs a `discrete_distribution` object with $n = 1$ and $p_0 = 1$. [*Note:* Such an object will always deliver the value 0. — *end note*]

```
template<class InputIterator>
  discrete_distribution(InputIterator firstW, InputIterator lastW);
```

4 *Requires:* `InputIterator` shall satisfy the requirements of an input iterator (Table 107) type. Moreover, `iterator_traits<InputIterator>::value_type` shall denote a type that is convertible to `double`. If `firstW == lastW`, let $n = 1$ and $w_0 = 1$. Otherwise, $\big[$`firstW`, `lastW`$\big)$ shall form a sequence w of length $n > 0$.

5 *Effects:* Constructs a `discrete_distribution` object with probabilities given by the formula above.

`discrete_distribution(initializer_list<double> wl);`

6 *Effects:* Same as `discrete_distribution(wl.begin(), wl.end())`.

```
template<class UnaryOperation>
  discrete_distribution(size_t nw, double xmin, double xmax, UnaryOperation fw);
```

7 *Requires:* Each instance of type `UnaryOperation` shall be a function object (20.283) whose return type shall be convertible to `double`. Moreover, `double` shall be convertible to the type of `UnaryOperation`'s sole parameter. If `nw` $= 0$, let $n = 1$, otherwise let $n =$ `nw`. The relation $0 < \delta = (\texttt{xmax} - \texttt{xmin})/n$ shall hold.

8 *Effects:* Constructs a `discrete_distribution` object with probabilities given by the formula above, using the following values: If `nw` $= 0$, let $w_0 = 1$. Otherwise, let $w_k = \texttt{fw}(\texttt{xmin} + k \cdot \delta + \delta/2)$ for $k = 0, \ldots, n-1$.

9 *Complexity:* The number of invocations of `fw` shall not exceed n.

`vector<double> probabilities() const;`

10 *Returns:* A `vector<double>` whose `size` member returns n and whose `operator[]` member returns p_k when invoked with argument k for $k = 0, \ldots, n-1$.

26.128.0.1.2 Class template `piecewise_constant_distribution` [rand.dist.samp.pconst]

1 A `piecewise_constant_distribution` random number distribution produces random numbers x, $b_0 \le x < b_n$, uniformly distributed over each subinterval $[b_i, b_{i+1})$ according to the probability density function

$$p(x \mid b_0, \ldots, b_n,\ \rho_0, \ldots, \rho_{n-1}) = \rho_i \text{ , for } b_i \le x < b_{i+1} \text{ .}$$

2 The $n+1$ distribution parameters b_i, also known as this distribution's *interval boundaries* , shall satisfy the relation $b_i < b_{i+1}$ for $i = 0, \ldots, n-1$. Unless specified otherwise, the remaining n distribution parameters are calculated as:

$$\rho_k = \frac{w_k}{S \cdot (b_{k+1} - b_k)} \text{ for } k = 0, \ldots, n-1,$$

in which the values w_k, commonly known as the *weights* , shall be non-negative, non-NaN, and non-infinity. Moreover, the following relation shall hold: $0 < S = w_0 + \cdots + w_{n-1}$.

```
template<class RealType = double>
 class piecewise_constant_distribution
{
public:
 // types
 typedef RealType result_type;
 typedef unspecified param_type;

 // constructor and reset functions
 piecewise_constant_distribution();
 template<class InputIteratorB, class InputIteratorW>
   piecewise_constant_distribution(InputIteratorB firstB, InputIteratorB lastB,
                                   InputIteratorW firstW);
 template<class UnaryOperation>
   piecewise_constant_distribution(initializer_list<RealType> bl, UnaryOperation fw);
 template<class UnaryOperation>
   piecewise_constant_distribution(size_t nw, RealType xmin, RealType xmax, UnaryOperation fw);
 explicit piecewise_constant_distribution(const param_type& parm);
 void reset();

 // generating functions
 template<class URNG>
   result_type operator()(URNG& g);
 template<class URNG>
   result_type operator()(URNG& g, const param_type& parm);

 // property functions
 vector<result_type> intervals() const;
 vector<result_type> densities() const;
 param_type param() const;
 void param(const param_type& parm);
 result_type min() const;
 result_type max() const;
};
```

`piecewise_constant_distribution();`

3 *Effects:* Constructs a `piecewise_constant_distribution` object with $n = 1$, $\rho_0 = 1$, $b_0 = 0$, and $b_1 = 1$.

```
template<class InputIteratorB, class InputIteratorW>
 piecewise_constant_distribution(InputIteratorB firstB, InputIteratorB lastB,
                                 InputIteratorW firstW);
```

4 *Requires:* `InputIteratorB` and `InputIteratorW` shall each satisfy the requirements of an input iterator (Table 107) type. Moreover, `iterator_traits<InputIteratorB>::value_type` and `iterator_traits<InputIteratorW>::value_type` shall each denote a type that is convertible to `double`. If `firstB == lastB` or `++firstB == lastB`, let $n = 1$, $w_0 = 1$, $b_0 = 0$, and $b_1 = 1$. Otherwise, $\big[$`firstB`, `lastB`$\big)$ shall form a sequence b of length $n+1$, the length of the sequence w starting from `firstW` shall be at least n, and any w_k for $k \geq n$ shall be ignored by the distribution.

5 *Effects:* Constructs a `piecewise_constant_distribution` object with parameters as specified above.

```
template<class UnaryOperation>
 piecewise_constant_distribution(initializer_list<RealType> bl, UnaryOperation fw);
```

6 *Requires:* Each instance of type `UnaryOperation` shall be a function object (20.283) whose return type shall be convertible to `double`. Moreover, `double` shall be convertible to the type of `UnaryOperation`'s sole parameter.

7 *Effects:* Constructs a `piecewise_constant_distribution` object with parameters taken or calculated from the following values: If `bl.size()` < 2, let $n = 1$, $w_0 = 1$, $b_0 = 0$, and $b_1 = 1$. Otherwise, let $\big[$`bl.begin()`, `bl.end()`$\big)$ form a sequence $b_0, \ldots, b_n$, and let $w_k =$ `fw`$\big((b_{k+1}+b_k)/2\big)$ for $k = 0, \ldots, n–1$.

8 *Complexity:* The number of invocations of `fw` shall not exceed n.

```
template<class UnaryOperation>
 piecewise_constant_distribution(size_t nw, RealType xmin, RealType xmax, UnaryOperation fw);
```

9 *Requires:* Each instance of type `UnaryOperation` shall be a function object (20.283) whose return type shall be convertible to `double`. Moreover, `double` shall be convertible to the type of `UnaryOperation`'s sole parameter. If `nw` $= 0$, let $n = 1$, otherwise let $n =$ `nw`. The relation $0 < \delta = (\texttt{xmax} - \texttt{xmin})/n$ shall hold.

10 *Effects:* Constructs a `piecewise_constant_distribution` object with parameters taken or calculated from the following values: Let $b_k =$ `xmin`$+k\cdot\delta$ for $k = 0, \ldots, n$, and $w_k =$ `fw`$(b_k+\delta/2)$ for $k = 0, \ldots, n–1$.

11 *Complexity:* The number of invocations of `fw` shall not exceed n.

```
vector<result_type> intervals() const;
```

12 *Returns:* A `vector<result_type>` whose `size` member returns $n+1$ and whose `operator[]` member returns b_k when invoked with argument k for $k = 0, \ldots, n$.

```
vector<result_type> densities() const;
```

13 *Returns:* A `vector<result_type>` whose `size` member returns n and whose `operator[]` member returns ρ_k when invoked with argument k for $k = 0, \ldots, n-1$.

26.128.0.1.3 Class template `piecewise_linear_distribution` [rand.dist.samp.plinear]

1 A `piecewise_linear_distribution` random number distribution produces random numbers x, $b_0 \leq x < b_n$, distributed over each subinterval $[b_i, b_{i+1})$ according to the probability density function

$$p(x \mid b_0, \ldots, b_n,\ \rho_0, \ldots, \rho_n) = \rho_i \cdot \frac{b_{i+1} - x}{b_{i+1} - b_i} + \rho_{i+1} \cdot \frac{x - b_i}{b_{i+1} - b_i}\ , \text{ for } b_i \leq x < b_{i+1}\ .$$

2 The $n+1$ distribution parameters b_i, also known as this distribution's *interval boundaries* , shall satisfy the relation $b_i < b_{i+1}$ for $i = 0, \ldots, n-1$. Unless specified otherwise, the remaining $n+1$ distribution parameters

are calculated as $\rho_k = w_k/S$ for $k = 0, \ldots, n$, in which the values w_k, commonly known as the *weights at boundaries* , shall be non-negative, non-NaN, and non-infinity. Moreover, the following relation shall hold:

$$0 < S = \frac{1}{2} \cdot \sum_{k=0}^{n-1} (w_k + w_{k+1}) \cdot (b_{k+1} - b_k) \ .$$

```
template<class RealType = double>
 class piecewise_linear_distribution
{
public:
 // types
 typedef RealType result_type;
 typedef unspecified param_type;

 // constructor and reset functions
 piecewise_linear_distribution();
 template<class InputIteratorB, class InputIteratorW>
   piecewise_linear_distribution(InputIteratorB firstB, InputIteratorB lastB,
                                 InputIteratorW firstW);
 template<class UnaryOperation>
   piecewise_linear_distribution(initializer_list<RealType> bl, UnaryOperation fw);
 template<class UnaryOperation>
   piecewise_linear_distribution(size_t nw, RealType xmin, RealType xmax, UnaryOperation fw);
 explicit piecewise_linear_distribution(const param_type& parm);
 void reset();

 // generating functions
 template<class URNG>
   result_type operator()(URNG& g);
 template<class URNG>
   result_type operator()(URNG& g, const param_type& parm);

 // property functions
 vector<result_type> intervals() const;
 vector<result_type> densities() const;
 param_type param() const;
 void param(const param_type& parm);
 result_type min() const;
 result_type max() const;
};
```

`piecewise_linear_distribution();`

3 *Effects:* Constructs a `piecewise_linear_distribution` object with $n = 1$, $\rho_0 = \rho_1 = 1$, $b_0 = 0$, and $b_1 = 1$.

```
template<class InputIteratorB, class InputIteratorW>
 piecewise_linear_distribution(InputIteratorB firstB, InputIteratorB lastB,
                               InputIteratorW firstW);
```

4 *Requires:* `InputIteratorB` and `InputIteratorW` shall each satisfy the requirements of an input iterator (Table 107) type. Moreover, `iterator_traits<InputIteratorB>::value_type` and `iterator_traits<InputIteratorW>::value_type` shall each denote a type that is convertible to `double`. If `firstB == lastB` or `++firstB == lastB`, let $n = 1$, $\rho_0 = \rho_1 = 1$, $b_0 = 0$, and $b_1 = 1$. Otherwise,

$\big[$`firstB`, `lastB`$\big)$ shall form a sequence b of length $n+1$, the length of the sequence w starting from `firstW` shall be at least $n+1$, and any w_k for $k \geq n+1$ shall be ignored by the distribution.

5 *Effects:* Constructs a `piecewise_linear_distribution` object with parameters as specified above.

```
template<class UnaryOperation>
 piecewise_linear_distribution(initializer_list<RealType> bl, UnaryOperation fw);
```

6 *Requires:* Each instance of type `UnaryOperation` shall be a function object (20.283) whose return type shall be convertible to `double`. Moreover, `double` shall be convertible to the type of `UnaryOperation`'s sole parameter.

7 *Effects:* Constructs a `piecewise_linear_distribution` object with parameters taken or calculated from the following values: If `bl.size()` < 2, let $n = 1$, $\rho_0 = \rho_1 = 1$, $b_0 = 0$, and $b_1 = 1$. Otherwise, let $\big[$`bl.begin()`,`bl.end()`$\big)$ form a sequence $b_0, \ldots, b_n$, and let $w_k =$ `fw`(b_k) for $k = 0, \ldots, n$.

8 *Complexity:* The number of invocations of `fw` shall not exceed $n+1$.

```
template<class UnaryOperation>
 piecewise_linear_distribution(size_t nw, RealType xmin, RealType xmax, UnaryOperation fw);
```

9 *Requires:* Each instance of type `UnaryOperation` shall be a function object (20.283) whose return type shall be convertible to `double`. Moreover, `double` shall be convertible to the type of `UnaryOperation`'s sole parameter. If `nw` $= 0$, let $n = 1$, otherwise let $n =$ `nw`. The relation $0 < \delta = (\texttt{xmax} - \texttt{xmin})/n$ shall hold.

10 *Effects:* Constructs a `piecewise_linear_distribution` object with parameters taken or calculated from the following values: Let $b_k = \texttt{xmin} + k \cdot \delta$ for $k = 0, \ldots, n$, and $w_k =$ `fw`(b_k) for $k = 0, \ldots, n$.

11 *Complexity:* The number of invocations of `fw` shall not exceed $n+1$.

```
vector<result_type> intervals() const;
```

12 *Returns:* A `vector<result_type>` whose `size` member returns $n+1$ and whose `operator[]` member returns b_k when invoked with argument k for $k = 0, \ldots, n$.

```
vector<result_type> densities() const;
```

13 *Returns:* A `vector<result_type>` whose `size` member returns n and whose `operator[]` member returns ρ_k when invoked with argument k for $k = 0, \ldots, n$.

26.129 Numeric arrays [numarray]

26.129.1 Header <valarray> synopsis [valarray.syn]

```
#include <initializer_list>

namespace std {

  template<class T> class valarray;          // An array of type T
  class slice;                               // a BLAS-like slice out of an array
  template<class T> class slice_array;
  class gslice;                              // a generalized slice out of an array
  template<class T> class gslice_array;
  template<class T> class mask_array;        // a masked array
  template<class T> class indirect_array;    // an indirected array

  template<class T> void swap(valarray<T>&, valarray<T>&) noexcept;

  template<class T> valarray<T> operator* (const valarray<T>&, const valarray<T>&);
```

```
template<class T> valarray<T> operator* (const valarray<T>&, const T&);
template<class T> valarray<T> operator* (const T&, const valarray<T>&);

template<class T> valarray<T> operator/ (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator/ (const valarray<T>&, const T&);
template<class T> valarray<T> operator/ (const T&, const valarray<T>&);

template<class T> valarray<T> operator% (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator% (const valarray<T>&, const T&);
template<class T> valarray<T> operator% (const T&, const valarray<T>&);

template<class T> valarray<T> operator+ (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator+ (const valarray<T>&, const T&);
template<class T> valarray<T> operator+ (const T&, const valarray<T>&);

template<class T> valarray<T> operator- (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator- (const valarray<T>&, const T&);
template<class T> valarray<T> operator- (const T&, const valarray<T>&);

template<class T> valarray<T> operator^ (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator^ (const valarray<T>&, const T&);
template<class T> valarray<T> operator^ (const T&, const valarray<T>&);

template<class T> valarray<T> operator& (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator& (const valarray<T>&, const T&);
template<class T> valarray<T> operator& (const T&, const valarray<T>&);

template<class T> valarray<T> operator| (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator| (const valarray<T>&, const T&);
template<class T> valarray<T> operator| (const T&, const valarray<T>&);

template<class T> valarray<T> operator<<(const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator<<(const valarray<T>&, const T&);
template<class T> valarray<T> operator<<(const T&, const valarray<T>&);

template<class T> valarray<T> operator>>(const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator>>(const valarray<T>&, const T&);
template<class T> valarray<T> operator>>(const T&, const valarray<T>&);

template<class T> valarray<bool> operator&&(const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator&&(const valarray<T>&, const T&);
template<class T> valarray<bool> operator&&(const T&, const valarray<T>&);

template<class T> valarray<bool> operator||(const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator||(const valarray<T>&, const T&);
template<class T> valarray<bool> operator||(const T&, const valarray<T>&);

template<class T>
  valarray<bool> operator==(const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator==(const valarray<T>&, const T&);
template<class T> valarray<bool> operator==(const T&, const valarray<T>&);
template<class T>
  valarray<bool> operator!=(const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator!=(const valarray<T>&, const T&);
template<class T> valarray<bool> operator!=(const T&, const valarray<T>&);
```

```
  template<class T>
    valarray<bool> operator< (const valarray<T>&, const valarray<T>&);
  template<class T> valarray<bool> operator< (const valarray<T>&, const T&);
  template<class T> valarray<bool> operator< (const T&, const valarray<T>&);
  template<class T>
    valarray<bool> operator> (const valarray<T>&, const valarray<T>&);
  template<class T> valarray<bool> operator> (const valarray<T>&, const T&);
  template<class T> valarray<bool> operator> (const T&, const valarray<T>&);
  template<class T>
    valarray<bool> operator<=(const valarray<T>&, const valarray<T>&);
  template<class T> valarray<bool> operator<=(const valarray<T>&, const T&);
  template<class T> valarray<bool> operator<=(const T&, const valarray<T>&);
  template<class T>
    valarray<bool> operator>=(const valarray<T>&, const valarray<T>&);
  template<class T> valarray<bool> operator>=(const valarray<T>&, const T&);
  template<class T> valarray<bool> operator>=(const T&, const valarray<T>&);

  template<class T> valarray<T> abs  (const valarray<T>&);
  template<class T> valarray<T> acos (const valarray<T>&);
  template<class T> valarray<T> asin (const valarray<T>&);
  template<class T> valarray<T> atan (const valarray<T>&);

  template<class T> valarray<T> atan2(const valarray<T>&, const valarray<T>&);
  template<class T> valarray<T> atan2(const valarray<T>&, const T&);
  template<class T> valarray<T> atan2(const T&, const valarray<T>&);

  template<class T> valarray<T> cos  (const valarray<T>&);
  template<class T> valarray<T> cosh (const valarray<T>&);
  template<class T> valarray<T> exp  (const valarray<T>&);
  template<class T> valarray<T> log  (const valarray<T>&);
  template<class T> valarray<T> log10(const valarray<T>&);

  template<class T> valarray<T> pow(const valarray<T>&, const valarray<T>&);
  template<class T> valarray<T> pow(const valarray<T>&, const T&);
  template<class T> valarray<T> pow(const T&, const valarray<T>&);

  template<class T> valarray<T> sin  (const valarray<T>&);
  template<class T> valarray<T> sinh (const valarray<T>&);
  template<class T> valarray<T> sqrt (const valarray<T>&);
  template<class T> valarray<T> tan  (const valarray<T>&);
  template<class T> valarray<T> tanh (const valarray<T>&);

  template <class T> unspecified1 begin(valarray<T>& v);
  template <class T> unspecified2 begin(const valarray<T>& v);
  template <class T> unspecified1 end(valarray<T>& v);
  template <class T> unspecified2 end(const valarray<T>& v);
}
```

1 The header `<valarray>` defines five class templates (`valarray`, `slice_array`, `gslice_array`, `mask_array`, and `indirect_array`), two classes (`slice` and `gslice`), and a series of related function templates for representing and manipulating arrays of values.

2

The `valarray` array classes are defined to be free of certain forms of aliasing, thus allowing operations on

these classes to be optimized.

1

Any function returning a `valarray<T>` is permitted to return an object of another type, provided all the const member functions of `valarray<T>` are also applicable to this type. This return type shall not add more than two levels of template nesting over the most deeply nested argument type.[280]

1

Implementations introducing such replacement types shall provide additional functions and operators as follows:

— for every function taking a `const valarray<T>&` other than `begin` and `end` (26.142.7), identical functions taking the replacement types shall be added;

— for every function taking two `const valarray<T>&` arguments, identical functions taking every combination of `const valarray<T>&` and replacement types shall be added.

1

In particular, an implementation shall allow a `valarray<T>` to be constructed from such replacement types and shall allow assignments and computed assignments of such types to `valarray<T>`, `slice_array<T>`, `gslice_array<T>`, `mask_array<T>` and `indirect_array<T>` objects.

1 These library functions are permitted to throw a `bad_alloc` (18.223.1.1) exception if there are not sufficient resources available to carry out the operation. Note that the exception is not mandated.

26.134.1 Class template `valarray` [template.valarray]

26.134.1.1 Class template `valarray` overview [template.valarray.overview]

```
namespace std {
  template<class T> class valarray {
  public:
    typedef T value_type;

    // 26.136.0.1 construct/destroy:
    valarray();
    explicit valarray(size_t);
    valarray(const T&, size_t);
    valarray(const T*, size_t);
    valarray(const valarray&);
    valarray(valarray&&) noexcept;
    valarray(const slice_array<T>&);
    valarray(const gslice_array<T>&);
    valarray(const mask_array<T>&);
    valarray(const indirect_array<T>&);
    valarray(initializer_list<T>);
    ~valarray();

    // 26.137.0.1 assignment:
    valarray<T>& operator=(const valarray<T>&);
    valarray<T>& operator=(valarray<T>&&) noexcept;
    valarray& operator=(initializer_list<T>);
    valarray<T>& operator=(const T&);
    valarray<T>& operator=(const slice_array<T>&);
```

280) Clause 18.41.2 recommends a minimum number of recursively nested template instantiations. This requirement thus indirectly suggests a minimum allowable complexity for valarray expressions.

```
valarray<T>& operator=(const gslice_array<T>&);
valarray<T>& operator=(const mask_array<T>&);
valarray<T>& operator=(const indirect_array<T>&);

// 26.139.0.1 element access:
const T&          operator[](size_t) const;
T&                operator[](size_t);

// 26.141.0.1 subset operations:
valarray<T>       operator[](slice) const;
slice_array<T>    operator[](slice);
valarray<T>       operator[](const gslice&) const;
gslice_array<T>   operator[](const gslice&);
valarray<T>       operator[](const valarray<bool>&) const;
mask_array<T>     operator[](const valarray<bool>&);
valarray<T>       operator[](const valarray<size_t>&) const;
indirect_array<T> operator[](const valarray<size_t>&);

// 26.141.0.2 unary operators:
valarray<T> operator+() const;
valarray<T> operator-() const;
valarray<T> operator~() const;
valarray<bool> operator!() const;

// 26.141.0.3 computed assignment:
valarray<T>& operator*= (const T&);
valarray<T>& operator/= (const T&);
valarray<T>& operator%= (const T&);
valarray<T>& operator+= (const T&);
valarray<T>& operator-= (const T&);
valarray<T>& operator^= (const T&);
valarray<T>& operator&= (const T&);
valarray<T>& operator|= (const T&);
valarray<T>& operator<<=(const T&);
valarray<T>& operator>>=(const T&);

valarray<T>& operator*= (const valarray<T>&);
valarray<T>& operator/= (const valarray<T>&);
valarray<T>& operator%= (const valarray<T>&);
valarray<T>& operator+= (const valarray<T>&);
valarray<T>& operator-= (const valarray<T>&);
valarray<T>& operator^= (const valarray<T>&);
valarray<T>& operator|= (const valarray<T>&);
valarray<T>& operator&= (const valarray<T>&);
valarray<T>& operator<<=(const valarray<T>&);
valarray<T>& operator>>=(const valarray<T>&);

// 26.141.0.4 member functions:
void swap(valarray&) noexcept;

size_t size() const;

T sum() const;
T min() const;
T max() const;
```

```
    valarray<T> shift (int) const;
    valarray<T> cshift(int) const;
    valarray<T> apply(T func(T)) const;
    valarray<T> apply(T func(const T&)) const;
    void resize(size_t sz, T c = T());
  };
}
```

1

The class template `valarray<T>` is a one-dimensional smart array, with elements numbered sequentially from zero. It is a representation of the mathematical concept of an ordered set of values. The illusion of higher dimensionality may be produced by the familiar idiom of computed indices, together with the powerful subsetting capabilities provided by the generalized subscript operators.[281]

1 An implementation is permitted to qualify any of the functions declared in `<valarray>` as `inline`.

26.136.0.1 `valarray` constructors [valarray.cons]

`valarray();`

1 *Effects:* Constructs an object of class `valarray<T>`[282] which has zero length.[283]

`explicit valarray(size_t);`

2 The array created by this constructor has a length equal to the value of the argument. The elements of the array are value-initialized (8.821).

`valarray(const T&, size_t);`

3 The array created by this constructor has a length equal to the second argument. The elements of the array are initialized with the value of the first argument.

`valarray(const T*, size_t);`

4

The array created by this constructor has a length equal to the second argument `n`. The values of the elements of the array are initialized with the first `n` values pointed to by the first argument.[284] If the value of the second argument is greater than the number of values pointed to by the first argument, the behavior is undefined.

`valarray(const valarray<T>&);`

1 The array created by this constructor has the same length as the argument array. The elements are initialized with the values of the corresponding elements of the argument array.[285]

`valarray(valarray<T>&& v) noexcept;`

281) The intent is to specify an array template that has the minimum functionality necessary to address aliasing ambiguities and the proliferation of temporaries. Thus, the `valarray` template is neither a matrix class nor a field class. However, it is a very useful building block for designing such classes.
282) For convenience, such objects are referred to as "arrays" throughout the remainder of 26.129.
283) This default constructor is essential, since arrays of `valarray` may be useful. After initialization, the length of an empty array can be increased with the `resize` member function.
284) This constructor is the preferred method for converting a C array to a `valarray` object.
285) This copy constructor creates a distinct array rather than an alias. Implementations in which arrays share storage are permitted, but they shall implement a copy-on-reference mechanism to ensure that arrays are conceptually distinct.

2 The array created by this constructor has the same length as the argument array. The elements are initialized with the values of the corresponding elements of the argument array.

3 *Complexity:* Constant.

```
valarray(initializer_list<T> il);
```

4 *Effects:* Same as `valarray(il.begin(), il.size())`.

```
valarray(const slice_array<T>&);
valarray(const gslice_array<T>&);
valarray(const mask_array<T>&);
valarray(const indirect_array<T>&);
```

5 These conversion constructors convert one of the four reference templates to a `valarray`.

```
~valarray();
```

6 The destructor is applied to every element of `*this`; an implementation may return all allocated memory.

26.137.0.1 `valarray` assignment [valarray.assign]

```
valarray<T>& operator=(const valarray<T>& v);
```

1

Each element of the `*this` array is assigned the value of the corresponding element of the argument array. If the length of `v` is not equal to the length of `*this` , resizes `*this` to make the two arrays the same length, as if by calling `resize(v.size())`, before performing the assignment.

1 *Postcondition:* `size() == v.size()`.

```
valarray<T>& operator=(valarray<T>&& v) noexcept;
```

2 *Effects:* `*this` obtains the value of `v`. The value of `v` after the assignment is not specified.

3 *Complexity:* Linear.

```
valarray& operator=(initializer_list<T> il);
```

4 *Effects:* `*this = valarray(il)`.

5 *Returns:* `*this`.

```
valarray<T>& operator=(const T&);
```

6 The scalar assignment operator causes each element of the `*this` array to be assigned the value of the argument.

```
valarray<T>& operator=(const slice_array<T>&);
valarray<T>& operator=(const gslice_array<T>&);
valarray<T>& operator=(const mask_array<T>&);
valarray<T>& operator=(const indirect_array<T>&);
```

7 *Requires:* The length of the array to which the argument refers equals `size()`.

8 These operators allow the results of a generalized subscripting operation to be assigned directly to a `valarray`.

9 If the value of an element in the left-hand side of a valarray assignment operator depends on the value of another element in that left-hand side, the resulting behavior is undefined.

26.139.0.1 `valarray` element access [valarray.access]

```
const T&  operator[](size_t) const;
T& operator[](size_t);
```

1 The subscript operator returns a reference to the corresponding element of the array.

2 Thus, the expression `(a[i] = q, a[i]) == q` evaluates as true for any non-constant `valarray<T> a`, any `T q`, and for any `size_t i` such that the value of `i` is less than the length of `a`.

3 The expression `&a[i+j] == &a[i] + j` evaluates as true for all `size_t i` and `size_t j` such that `i+j` is less than the length of the array `a`.

1 Likewise, the expression `&a[i] != &b[j]` evaluates as `true` for any two arrays `a` and `b` and for any `size_t i` and `size_t j` such that `i` is less than the length of `a` and `j` is less than the length of `b`. This property indicates an absence of aliasing and may be used to advantage by optimizing compilers.[286]

1 The reference returned by the subscript operator for an array shall be valid until the member function `resize(size_t, T)` (26.141.0.4) is called for that array or until the lifetime of that array ends, whichever happens first.

2 If the subscript operator is invoked with a `size_t` argument whose value is not less than the length of the array, the behavior is undefined.

26.141.0.1 `valarray` subset operations [valarray.sub]

1 The member `operator[]` is overloaded to provide several ways to select sequences of elements from among those controlled by `*this`. Each of these operations returns a subset of the array. The const-qualified versions return this subset as a new `valarray` object. The non-const versions return a class template object which has reference semantics to the original array, working in conjunction with various overloads of `operator=` and other assigning operators to allow selective replacement (slicing) of the controlled sequence. In each case the selected element(s) must exist.

```
valarray<T> operator[](slice slicearr) const;
```

2 *Returns:* An object of class `valarray<T>` containing those elements of the controlled sequence designated by `slicearr`. [*Example:*

```
const valarray<char> v0("abcdefghijklmnop", 16);
// v0[slice(2, 5, 3)] returns valarray<char>("cfilo", 5)
```

— *end example*]

```
slice_array<T> operator[](slice slicearr);
```

3 *Returns:* An object that holds references to elements of the controlled sequence selected by `slicearr`. [*Example:*

```
valarray<char> v0("abcdefghijklmnop", 16);
valarray<char> v1("ABCDE", 5);
v0[slice(2, 5, 3)] = v1;
// v0 == valarray<char>("abAdeBghCjkDmnEp", 16);
```

— *end example*]

286) Compilers may take advantage of inlining, constant propagation, loop fusion, tracking of pointers obtained from `operator new`, and other techniques to generate efficient `valarrays`.

```
valarray<T> operator[](const gslice& gslicearr) const;
```

4 *Returns:* An object of class `valarray<T>` containing those elements of the controlled sequence designated by `gslicearr`. [*Example:*

```
const valarray<char> v0("abcdefghijklmnop", 16);
const size_t lv[] = { 2, 3 };
const size_t dv[] = { 7, 2 };
const valarray<size_t> len(lv, 2), str(dv, 2);
// v0[gslice(3, len, str)] returns
// valarray<char>("dfhkmo", 6)
```

— *end example*]

```
gslice_array<T> operator[](const gslice& gslicearr);
```

5 *Returns:* An object that holds references to elements of the controlled sequence selected by `gslicearr`. [*Example:*

```
valarray<char> v0("abcdefghijklmnop", 16);
valarray<char> v1("ABCDE", 5);
const size_t lv[] = { 2, 3 };
const size_t dv[] = { 7, 2 };
const valarray<size_t> len(lv, 2), str(dv, 2);
v0[gslice(3, len, str)] = v1;
// v0 == valarray<char>("abcAeBgCijDlEnFp", 16)
```

— *end example*]

```
valarray<T> operator[](const valarray<bool>& boolarr) const;
```

6 *Returns:* An object of class `valarray<T>` containing those elements of the controlled sequence designated by `boolarr`. [*Example:*

```
const valarray<char> v0("abcdefghijklmnop", 16);
const bool vb[] = { false, false, true, true, false, true };
// v0[valarray<bool>(vb, 6)] returns
// valarray<char>("cdf", 3)
```

— *end example*]

```
mask_array<T> operator[](const valarray<bool>& boolarr);
```

7 *Returns:* An object that holds references to elements of the controlled sequence selected by `boolarr`. [*Example:*

```
valarray<char> v0("abcdefghijklmnop", 16);
valarray<char> v1("ABC", 3);
const bool vb[] = { false, false, true, true, false, true };
v0[valarray<bool>(vb, 6)] = v1;
// v0 == valarray<char>("abABeCghijklmnop", 16)
```

— *end example*]

```
valarray<T> operator[](const valarray<size_t>& indarr) const;
```

8 *Returns:* An object of class `valarray<T>` containing those elements of the controlled sequence designated by `indarr`. [*Example:*

```
const valarray<char> v0("abcdefghijklmnop", 16);
const size_t vi[] = { 7, 5, 2, 3, 8 };
// v0[valarray<size_t>(vi, 5)] returns
// valarray<char>("hfcdi", 5)
```

— *end example*]

```
indirect_array<T> operator[](const valarray<size_t>& indarr);
```

9 *Returns:* An object that holds references to elements of the controlled sequence selected by `indarr`. [*Example:*

```
valarray<char> v0("abcdefghijklmnop", 16);
valarray<char> v1("ABCDE", 5);
const size_t vi[] = { 7, 5, 2, 3, 8 };
v0[valarray<size_t>(vi, 5)] = v1;
// v0 == valarray<char>("abCDeBgAEjklmnop", 16)
```

— *end example*]

26.141.0.2 `valarray` unary operators [valarray.unary]

```
valarray<T> operator+() const;
valarray<T> operator-() const;
valarray<T> operator~() const;
valarray<bool> operator!() const;
```

1 Each of these operators may only be instantiated for a type `T` to which the indicated operator can be applied and for which the indicated operator returns a value which is of type `T` (`bool` for `operator!`) or which may be unambiguously implicitly converted to type `T` (`bool` for `operator!`).

2 Each of these operators returns an array whose length is equal to the length of the array. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding element of the array.

26.141.0.3 `valarray` computed assignment [valarray.cassign]

```
valarray<T>& operator*= (const valarray<T>&);
valarray<T>& operator/= (const valarray<T>&);
valarray<T>& operator%= (const valarray<T>&);
valarray<T>& operator+= (const valarray<T>&);
valarray<T>& operator-= (const valarray<T>&);
valarray<T>& operator^= (const valarray<T>&);
valarray<T>& operator&= (const valarray<T>&);
valarray<T>& operator|= (const valarray<T>&);
valarray<T>& operator<<=(const valarray<T>&);
valarray<T>& operator>>=(const valarray<T>&);
```

1 Each of these operators may only be instantiated for a type `T` to which the indicated operator can be applied. Each of these operators performs the indicated operation on each of its elements and the corresponding element of the argument array.

2 The array is then returned by reference.

3 If the array and the argument array do not have the same length, the behavior is undefined. The appearance of an array on the left-hand side of a computed assignment does `not` invalidate references or pointers.

4 If the value of an element in the left-hand side of a valarray computed assignment operator depends on the value of another element in that left hand side, the resulting behavior is undefined.

```
valarray<T>& operator*= (const T&);
valarray<T>& operator/= (const T&);
valarray<T>& operator%= (const T&);
valarray<T>& operator+= (const T&);
valarray<T>& operator-= (const T&);
valarray<T>& operator^= (const T&);
valarray<T>& operator&= (const T&);
valarray<T>& operator|= (const T&);
valarray<T>& operator<<=(const T&);
valarray<T>& operator>>=(const T&);
```

5 Each of these operators may only be instantiated for a type `T` to which the indicated operator can be applied.

6 Each of these operators applies the indicated operation to each element of the array and the non-array argument.

7 The array is then returned by reference.

8 The appearance of an array on the left-hand side of a computed assignment does *not* invalidate references or pointers to the elements of the array.

26.141.0.4 `valarray` member functions [valarray.members]

```
void swap(valarray& v) noexcept;
```

1 *Effects:* `*this` obtains the value of `v`. `v` obtains the value of `*this`.

2 *Complexity:* Constant.

```
size_t size() const;
```

3 *Returns:* The number of elements in the array.

4 *Complexity:* Constant time.

```
T sum() const;
```

This function may only be instantiated for a type `T` to which `operator+=` can be applied. This function returns the sum of all the elements of the array.

5 If the array has length 0, the behavior is undefined. If the array has length 1, `sum()` returns the value of element 0. Otherwise, the returned value is calculated by applying `operator+=` to a copy of an element of the array and all other elements of the array in an unspecified order.

```
T min() const;
```

6 This function returns the minimum value contained in `*this`. The value returned for an array of length 0 is undefined. For an array of length 1, the value of element 0 is returned. For all other array lengths, the determination is made using `operator<`.

```
T max() const;
```

7 This function returns the maximum value contained in `*this`. The value returned for an array of length 0 is undefined. For an array of length 1, the value of element 0 is returned. For all other array lengths, the determination is made using `operator<`.

```
valarray<T> shift(int n) const;
```

8 This function returns an object of class `valarray<T>` of length `size()`, each of whose elements I is `(*this)[I + n]` if `I + n` is non-negative and less than `size()`, otherwise `T()`. Thus if element zero is taken as the leftmost element, a positive value of `n` shifts the elements left `n` places, with zero fill.

9 [*Example:* If the argument has the value -2, the first two elements of the result will be value-initialized (8.821); the third element of the result will be assigned the value of the first element of the argument; etc. — *end example*]

```
valarray<T> cshift(int n) const;
```

10 This function returns an object of class `valarray<T>` of length `size()` that is a circular shift of `*this`. If element zero is taken as the leftmost element, a non-negative value of n shifts the elements circularly left n places and a negative value of n shifts the elements circularly right -n places.

```
valarray<T> apply(T func(T)) const;
valarray<T> apply(T func(const T&)) const;
```

11 These functions return an array whose length is equal to the array. Each element of the returned array is assigned the value returned by applying the argument function to the corresponding element of the array.

```
void resize(size_t sz, T c = T());
```

12 This member function changes the length of the `*this` array to `sz` and then assigns to each element the value of the second argument. Resizing invalidates all pointers and references to elements in the array.

26.141.1 `valarray` non-member operations [valarray.nonmembers]

26.141.1.1 `valarray` binary operators [valarray.binary]

```
template<class T> valarray<T> operator*
    (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator/
    (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator%
    (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator+
    (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator-
    (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator^
    (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator&
    (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator|
    (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator<<
    (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator>>
    (const valarray<T>&, const valarray<T>&);
```

1 Each of these operators may only be instantiated for a type `T` to which the indicated operator can be applied and for which the indicated operator returns a value which is of type `T` or which can be unambiguously implicitly converted to type `T`.

2 Each of these operators returns an array whose length is equal to the lengths of the argument arrays. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding elements of the argument arrays.

3 If the argument arrays do not have the same length, the behavior is undefined.

```
template<class T> valarray<T> operator* (const valarray<T>&, const T&);
template<class T> valarray<T> operator* (const T&, const valarray<T>&);
template<class T> valarray<T> operator/ (const valarray<T>&, const T&);
template<class T> valarray<T> operator/ (const T&, const valarray<T>&);
template<class T> valarray<T> operator% (const valarray<T>&, const T&);
template<class T> valarray<T> operator% (const T&, const valarray<T>&);
template<class T> valarray<T> operator+ (const valarray<T>&, const T&);
template<class T> valarray<T> operator+ (const T&, const valarray<T>&);
template<class T> valarray<T> operator- (const valarray<T>&, const T&);
template<class T> valarray<T> operator- (const T&, const valarray<T>&);
template<class T> valarray<T> operator^ (const valarray<T>&, const T&);
template<class T> valarray<T> operator^ (const T&, const valarray<T>&);
template<class T> valarray<T> operator& (const valarray<T>&, const T&);
template<class T> valarray<T> operator& (const T&, const valarray<T>&);
template<class T> valarray<T> operator| (const valarray<T>&, const T&);
template<class T> valarray<T> operator| (const T&, const valarray<T>&);
template<class T> valarray<T> operator<<(const valarray<T>&, const T&);
template<class T> valarray<T> operator<<(const T&, const valarray<T>&);
template<class T> valarray<T> operator>>(const valarray<T>&, const T&);
template<class T> valarray<T> operator>>(const T&, const valarray<T>&);
```

4 Each of these operators may only be instantiated for a type `T` to which the indicated operator can be applied and for which the indicated operator returns a value which is of type `T` or which can be unambiguously implicitly converted to type `T`.

5 Each of these operators returns an array whose length is equal to the length of the array argument. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding element of the array argument and the non-array argument.

26.141.1.2 `valarray` logical operators [valarray.comparison]

```
template<class T> valarray<bool> operator==
    (const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator!=
    (const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator<
    (const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator>
    (const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator<=
    (const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator>=
    (const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator&&
    (const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator||
    (const valarray<T>&, const valarray<T>&);
```

1 Each of these operators may only be instantiated for a type `T` to which the indicated operator can be applied and for which the indicated operator returns a value which is of type `bool` or which can be unambiguously implicitly converted to type `bool`.

2 Each of these operators returns a `bool` array whose length is equal to the length of the array arguments. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding elements of the argument arrays.

3 If the two array arguments do not have the same length, the behavior is undefined.

```
template<class T> valarray<bool> operator==(const valarray<T>&, const T&);
template<class T> valarray<bool> operator==(const T&, const valarray<T>&);
template<class T> valarray<bool> operator!=(const valarray<T>&, const T&);
template<class T> valarray<bool> operator!=(const T&, const valarray<T>&);
template<class T> valarray<bool> operator< (const valarray<T>&, const T&);
template<class T> valarray<bool> operator< (const T&, const valarray<T>&);
template<class T> valarray<bool> operator> (const valarray<T>&, const T&);
template<class T> valarray<bool> operator> (const T&, const valarray<T>&);
template<class T> valarray<bool> operator<=(const valarray<T>&, const T&);
template<class T> valarray<bool> operator<=(const T&, const valarray<T>&);
template<class T> valarray<bool> operator>=(const valarray<T>&, const T&);
template<class T> valarray<bool> operator>=(const T&, const valarray<T>&);
template<class T> valarray<bool> operator&&(const valarray<T>&, const T&);
template<class T> valarray<bool> operator&&(const T&, const valarray<T>&);
template<class T> valarray<bool> operator||(const valarray<T>&, const T&);
template<class T> valarray<bool> operator||(const T&, const valarray<T>&);
```

4 Each of these operators may only be instantiated for a type `T` to which the indicated operator can be applied and for which the indicated operator returns a value which is of type `bool` or which can be unambiguously implicitly converted to type `bool`.

5 Each of these operators returns a `bool` array whose length is equal to the length of the array argument. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding element of the array and the non-array argument.

26.141.1.3 `valarray` transcendentals [valarray.transcend]

```
template<class T> valarray<T> abs  (const valarray<T>&);
template<class T> valarray<T> acos (const valarray<T>&);
template<class T> valarray<T> asin (const valarray<T>&);
template<class T> valarray<T> atan (const valarray<T>&);
template<class T> valarray<T> atan2
    (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> atan2(const valarray<T>&, const T&);
template<class T> valarray<T> atan2(const T&, const valarray<T>&);
template<class T> valarray<T> cos  (const valarray<T>&);
template<class T> valarray<T> cosh (const valarray<T>&);
template<class T> valarray<T> exp  (const valarray<T>&);
template<class T> valarray<T> log  (const valarray<T>&);
template<class T> valarray<T> log10(const valarray<T>&);
template<class T> valarray<T> pow
    (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> pow  (const valarray<T>&, const T&);
template<class T> valarray<T> pow  (const T&, const valarray<T>&);
template<class T> valarray<T> sin  (const valarray<T>&);
template<class T> valarray<T> sinh (const valarray<T>&);
template<class T> valarray<T> sqrt (const valarray<T>&);
template<class T> valarray<T> tan  (const valarray<T>&);
template<class T> valarray<T> tanh (const valarray<T>&);
```

1

Each of these functions may only be instantiated for a type `T` to which a unique function with the indicated name can be applied (unqualified). This function shall return a value which is of type `T` or which can be unambiguously implicitly converted to type `T`.

26.142.0.1 `valarray` specialized algorithms [valarray.special]

```
template <class T> void swap(valarray<T>& x, valarray<T>& y) noexcept;
```

1 *Effects:* `x.swap(y)`.

26.142.1 Class `slice` [class.slice]

26.142.1.1 Class `slice` overview [class.slice.overview]

```
namespace std {
  class slice {
  public:
    slice();
    slice(size_t, size_t, size_t);

    size_t start() const;
    size_t size() const;
    size_t stride() const;
  };
}
```

1 The `slice` class represents a BLAS-like slice from an array. Such a slice is specified by a starting index, a length, and a stride.[287]

26.142.1.2 `slice` constructors [cons.slice]

```
slice();
slice(size_t start, size_t length, size_t stride);
slice(const slice&);
```

1 The default constructor is equivalent to `slice(0, 0, 0)`. A default constructor is provided only to permit the declaration of arrays of slices. The constructor with arguments for a slice takes a start, length, and stride parameter.

2 [*Example:* `slice(3, 8, 2)` constructs a slice which selects elements 3, 5, 7, ... 17 from an array. — *end example*]

26.142.1.3 `slice` access functions [slice.access]

```
size_t start() const;
size_t size() const;
size_t stride() const;
```

1 *Returns:* The start, length, or stride specified by a `slice` object.

2 *Complexity:* Constant time.

26.142.2 Class template `slice_array` [template.slice.array]

26.142.2.1 Class template `slice_array` overview [template.slice.array.overview]

```
namespace std {
  template <class T> class slice_array {
  public:
    typedef T value_type;

    void operator=  (const valarray<T>&) const;
```

287) BLAS stands for *Basic Linear Algebra Subprograms.* C++ programs may instantiate this class. See, for example, Dongarra, Du Croz, Duff, and Hammerling: *A set of Level 3 Basic Linear Algebra Subprograms*; Technical Report MCS-P1-0888, Argonne National Laboratory (USA), Mathematics and Computer Science Division, August, 1988.

```
    void operator*= (const valarray<T>&) const;
    void operator/= (const valarray<T>&) const;
    void operator%= (const valarray<T>&) const;
    void operator+= (const valarray<T>&) const;
    void operator-= (const valarray<T>&) const;
    void operator^= (const valarray<T>&) const;
    void operator&= (const valarray<T>&) const;
    void operator|= (const valarray<T>&) const;
    void operator<<=(const valarray<T>&) const;
    void operator>>=(const valarray<T>&) const;

    slice_array(const slice_array&);
    ~slice_array();
    const slice_array& operator=(const slice_array&) const;
  void operator=(const T&) const;

    slice_array() = delete;        // as implied by declaring copy constructor above
  };
}
```

1 The `slice_array` template is a helper template used by the `slice` subscript operator

```
slice_array<T> valarray<T>::operator[](slice);
```

It has reference semantics to a subset of an array specified by a `slice` object.

2 [*Example:* The expression `a[slice(1, 5, 3)] = b;` has the effect of assigning the elements of `b` to a slice of the elements in `a`. For the slice shown, the elements selected from `a` are 1, 4, ..., 13. — *end example*]

26.142.2.2 `slice_array` assignment [slice.arr.assign]

```
void operator=(const valarray<T>&) const;
const slice_array& operator=(const slice_array&) const;
```

1 These assignment operators have reference semantics, assigning the values of the argument array elements to selected elements of the `valarray<T>` object to which the `slice_array` object refers.

26.142.2.3 `slice_array` computed assignment [slice.arr.comp.assign]

```
void operator*= (const valarray<T>&) const;
void operator/= (const valarray<T>&) const;
void operator%= (const valarray<T>&) const;
void operator+= (const valarray<T>&) const;
void operator-= (const valarray<T>&) const;
void operator^= (const valarray<T>&) const;
void operator&= (const valarray<T>&) const;
void operator|= (const valarray<T>&) const;
void operator<<=(const valarray<T>&) const;
void operator>>=(const valarray<T>&) const;
```

1 These computed assignments have reference semantics, applying the indicated operation to the elements of the argument array and selected elements of the `valarray<T>` object to which the `slice_array` object refers.

26.142.2.4 `slice_array` fill function [slice.arr.fill]

```
void operator=(const T&) const;
```

1 This function has reference semantics, assigning the value of its argument to the elements of the `valarray<T>` object to which the `slice_array` object refers.

26.142.3 The `gslice` class [class.gslice]

26.142.3.1 The `gslice` class overview [class.gslice.overview]

```
namespace std {
  class gslice {
  public:
    gslice();
    gslice(size_t s, const valarray<size_t>& l, const valarray<size_t>& d);

    size_t           start() const;
    valarray<size_t> size() const;
    valarray<size_t> stride() const;
  };
}
```

1 This class represents a generalized slice out of an array. A `gslice` is defined by a starting offset (s), a set of lengths (l_j), and a set of strides (d_j). The number of lengths shall equal the number of strides.

2 A `gslice` represents a mapping from a set of indices (i_j), equal in number to the number of strides, to a single index k. It is useful for building multidimensional array classes using the `valarray` template, which is one-dimensional. The set of one-dimensional index values specified by a `gslice` are

$$k = s + \sum_j i_j d_j$$

where the multidimensional indices i_j range in value from 0 to $l_{ij} - 1$.

3 [*Example:* The `gslice` specification

```
start  = 3
length = {2, 4, 3}
stride = {19, 4, 1}
```

yields the sequence of one-dimensional indices

$$k = 3 + (0,1) \times 19 + (0,1,2,3) \times 4 + (0,1,2) \times 1$$

which are ordered as shown in the following table:

$(i_0,$	$i_1,$	$i_2,$	$k)$	$=$
	(0,	0,	0,	3),
	(0,	0,	1,	4),
	(0,	0,	2,	5),
	(0,	1,	0,	7),
	(0,	1,	1,	8),
	(0,	1,	2,	9),
	(0,	2,	0,	11),
	(0,	2,	1,	12),
	(0,	2,	2,	13),
	(0,	3,	0,	15),
	(0,	3,	1,	16),
	(0,	3,	2,	17),

(1, 0, 0, 22),
(1, 0, 1, 23),
...
(1, 3, 2, 36)

That is, the highest-ordered index turns fastest. — *end example*]

4 It is possible to have degenerate generalized slices in which an address is repeated.

5 [*Example:* If the stride parameters in the previous example are changed to {1, 1, 1}, the first few elements of the resulting sequence of indices will be

(0, 0, 0, 3),
(0, 0, 1, 4),
(0, 0, 2, 5),
(0, 1, 0, 4),
(0, 1, 1, 5),
(0, 1, 2, 6),
...

— *end example*]

6 If a degenerate slice is used as the argument to the non-`const` version of `operator[](const gslice&)`, the resulting behavior is undefined.

26.142.3.2 `gslice` constructors [gslice.cons]

```
gslice();
gslice(size_t start, const valarray<size_t>& lengths,
         const valarray<size_t>& strides);
gslice(const gslice&);
```

1 The default constructor is equivalent to `gslice(0, valarray<size_t>(), valarray<size_t>())`. The constructor with arguments builds a `gslice` based on a specification of start, lengths, and strides, as explained in the previous section.

26.142.3.3 `gslice` access functions [gslice.access]

```
size_t           start()  const;
valarray<size_t> size() const;
valarray<size_t> stride() const;
```

1 *Returns:* The representation of the start, lengths, or strides specified for the `gslice`.

2 *Complexity:* `start()` is constant time. `size()` and `stride()` are linear in the number of strides.

26.142.4 Class template `gslice_array` [template.gslice.array]

26.142.4.1 Class template `gslice_array` overview [template.gslice.array.overview]

```
namespace std {
  template <class T> class gslice_array {
  public:
    typedef T value_type;

    void operator=  (const valarray<T>&) const;
    void operator*= (const valarray<T>&) const;
    void operator/= (const valarray<T>&) const;
    void operator%= (const valarray<T>&) const;
    void operator+= (const valarray<T>&) const;
```

```
    void operator-= (const valarray<T>&) const;
    void operator^= (const valarray<T>&) const;
    void operator&= (const valarray<T>&) const;
    void operator|= (const valarray<T>&) const;
    void operator<<=(const valarray<T>&) const;
    void operator>>=(const valarray<T>&) const;

    gslice_array(const gslice_array&);
    ~gslice_array();
    const gslice_array& operator=(const gslice_array&) const;
    void operator=(const T&) const;

    gslice_array() = delete;       // as implied by declaring copy constructor above
  };
}
```

1 This template is a helper template used by the `slice` subscript operator

```
gslice_array<T> valarray<T>::operator[](const gslice&);
```

2 It has reference semantics to a subset of an array specified by a `gslice` object.

3 Thus, the expression `a[gslice(1, length, stride)] = b` has the effect of assigning the elements of `b` to a generalized slice of the elements in `a`.

26.142.4.2 `gslice_array` assignment [gslice.array.assign]

```
void operator=(const valarray<T>&) const;
const gslice_array& operator=(const gslice_array&) const;
```

1 These assignment operators have reference semantics, assigning the values of the argument array elements to selected elements of the `valarray<T>` object to which the `gslice_array` refers.

26.142.4.3 `gslice_array` [gslice.array.comp.assign]

```
void operator*= (const valarray<T>&) const;
void operator/= (const valarray<T>&) const;
void operator%= (const valarray<T>&) const;
void operator+= (const valarray<T>&) const;
void operator-= (const valarray<T>&) const;
void operator^= (const valarray<T>&) const;
void operator&= (const valarray<T>&) const;
void operator|= (const valarray<T>&) const;
void operator<<=(const valarray<T>&) const;
void operator>>=(const valarray<T>&) const;
```

1 These computed assignments have reference semantics, applying the indicated operation to the elements of the argument array and selected elements of the `valarray<T>` object to which the `gslice_array` object refers.

26.142.4.4 `gslice_array` fill function [gslice.array.fill]

```
void operator=(const T&) const;
```

1 This function has reference semantics, assigning the value of its argument to the elements of the `valarray<T>` object to which the `gslice_array` object refers.

26.142.5 Class template `mask_array` [template.mask.array]

26.142.5.1 Class template `mask_array` overview [template.mask.array.overview]

```
namespace std {
  template <class T> class mask_array {
  public:
    typedef T value_type;

    void operator=  (const valarray<T>&) const;
    void operator*= (const valarray<T>&) const;
    void operator/= (const valarray<T>&) const;
    void operator%= (const valarray<T>&) const;
    void operator+= (const valarray<T>&) const;
    void operator-= (const valarray<T>&) const;
    void operator^= (const valarray<T>&) const;
    void operator&= (const valarray<T>&) const;
    void operator|= (const valarray<T>&) const;
    void operator<<=(const valarray<T>&) const;
    void operator>>=(const valarray<T>&) const;

    mask_array(const mask_array&);
   ~mask_array();
    const mask_array& operator=(const mask_array&) const;
    void operator=(const T&) const;

    mask_array() = delete;        // as implied by declaring copy constructor above
  };
}
```

1 This template is a helper template used by the mask subscript operator:

`mask_array<T> valarray<T>::operator[](const valarray<bool>&).`

2 It has reference semantics to a subset of an array specified by a boolean mask. Thus, the expression `a[mask] = b;` has the effect of assigning the elements of `b` to the masked elements in `a` (those for which the corresponding element in `mask` is `true`.)

26.142.5.2 `mask_array` assignment [mask.array.assign]

```
void operator=(const valarray<T>&) const;
const mask_array& operator=(const mask_array&) const;
```

1 These assignment operators have reference semantics, assigning the values of the argument array elements to selected elements of the `valarray<T>` object to which it refers.

26.142.5.3 `mask_array` computed assignment [mask.array.comp.assign]

```
void operator*= (const valarray<T>&) const;
void operator/= (const valarray<T>&) const;
void operator%= (const valarray<T>&) const;
void operator+= (const valarray<T>&) const;
void operator-= (const valarray<T>&) const;
void operator^= (const valarray<T>&) const;
void operator&= (const valarray<T>&) const;
void operator|= (const valarray<T>&) const;
void operator<<=(const valarray<T>&) const;
void operator>>=(const valarray<T>&) const;
```

1 These computed assignments have reference semantics, applying the indicated operation to the elements of the argument array and selected elements of the `valarray<T>` object to which the mask object refers.

26.142.5.4 `mask_array` fill function [mask.array.fill]

```
void operator=(const T&) const;
```

1 This function has reference semantics, assigning the value of its argument to the elements of the `valarray<T>` object to which the `mask_array` object refers.

26.142.6 Class template `indirect_array` [template.indirect.array]

26.142.6.1 Class template `indirect_array` overview [template.indirect.array.overview]

```
namespace std {
  template <class T> class indirect_array {
  public:
    typedef T value_type;

    void operator=  (const valarray<T>&) const;
    void operator*= (const valarray<T>&) const;
    void operator/= (const valarray<T>&) const;
    void operator%= (const valarray<T>&) const;
    void operator+= (const valarray<T>&) const;
    void operator-= (const valarray<T>&) const;
    void operator^= (const valarray<T>&) const;
    void operator&= (const valarray<T>&) const;
    void operator|= (const valarray<T>&) const;
    void operator<<=(const valarray<T>&) const;
    void operator>>=(const valarray<T>&) const;

    indirect_array(const indirect_array&);
   ~indirect_array();
    const indirect_array& operator=(const indirect_array&) const;
    void operator=(const T&) const;

    indirect_array() = delete;        // as implied by declaring copy constructor above
  };
}
```

1 This template is a helper template used by the indirect subscript operator

`indirect_array<T> valarray<T>::operator[](const valarray<size_t>&).`

2 It has reference semantics to a subset of an array specified by an `indirect_array`. Thus the expression `a[indirect] = b;` has the effect of assigning the elements of `b` to the elements in `a` whose indices appear in `indirect`.

26.142.6.2 `indirect_array` assignment [indirect.array.assign]

```
void operator=(const valarray<T>&) const;
const indirect_array& operator=(const indirect_array&) const;
```

1 These assignment operators have reference semantics, assigning the values of the argument array elements to selected elements of the `valarray<T>` object to which it refers.

2 If the `indirect_array` specifies an element in the `valarray<T>` object to which it refers more than once, the behavior is undefined.

3 [*Example:*

```
int addr[] = {2, 3, 1, 4, 4};
valarray<size_t> indirect(addr, 5);
```

```
valarray<double> a(0., 10), b(1., 5);
a[indirect] = b;
```

results in undefined behavior since element 4 is specified twice in the indirection. — *end example*]

26.142.6.3 `indirect_array` computed assignment [indirect.array.comp.assign]

```
void operator*= (const valarray<T>&) const;
void operator/= (const valarray<T>&) const;
void operator%= (const valarray<T>&) const;
void operator+= (const valarray<T>&) const;
void operator-= (const valarray<T>&) const;
void operator^= (const valarray<T>&) const;
void operator&= (const valarray<T>&) const;
void operator|= (const valarray<T>&) const;
void operator<<=(const valarray<T>&) const;
void operator>>=(const valarray<T>&) const;
```

1 These computed assignments have reference semantics, applying the indicated operation to the elements of the argument array and selected elements of the `valarray<T>` object to which the `indirect_array` object refers.

2 If the `indirect_array` specifies an element in the `valarray<T>` object to which it refers more than once, the behavior is undefined.

26.142.6.4 `indirect_array` fill function [indirect.array.fill]

```
void operator=(const T&) const;
```

1 This function has reference semantics, assigning the value of its argument to the elements of the `valarray<T>` object to which the `indirect_array` object refers.

26.142.7 valarray range access [valarray.range]

1

In the `begin` and `end` function templates that follow, *`unspecified`*1 is a type that meets the requirements of a mutable random access iterator (24.167.1) whose `value_type` is the template parameter `T` and whose `reference` type is `T&`. *`unspecified`*2 is a type that meets the requirements of a constant random access iterator (24.167.1) whose `value_type` is the template parameter `T` and whose `reference` type is `const T&`.

1 The iterators returned by `begin` and `end` for an array are guaranteed to be valid until the member function `resize(size_t, T)` (26.141.0.4) is called for that array or until the lifetime of that array ends, whichever happens first.

```
template <class T> unspecified1 begin(valarray<T>& v);
template <class T> unspecified2 begin(const valarray<T>& v);
```

2 *Returns:* An iterator referencing the first value in the numeric array.

```
template <class T> unspecified1 end(valarray<T>& v);
template <class T> unspecified2 end(const valarray<T>& v);
```

3 *Returns:* An iterator referencing one past the last value in the numeric array.

26.144 Generalized numeric operations [numeric.ops]

26.144.1 Header `<numeric>` synopsis [numeric.ops.overview]

```
namespace std {
  template <class InputIterator, class T>
    T accumulate(InputIterator first, InputIterator last, T init);
  template <class InputIterator, class T, class BinaryOperation>
    T accumulate(InputIterator first, InputIterator last, T init,
                 BinaryOperation binary_op);

  template <class InputIterator1, class InputIterator2, class T>
    T inner_product(InputIterator1 first1, InputIterator1 last1,
                    InputIterator2 first2, T init);
  template <class InputIterator1, class InputIterator2, class T,
            class BinaryOperation1, class BinaryOperation2>
    T inner_product(InputIterator1 first1, InputIterator1 last1,
                    InputIterator2 first2, T init,
                    BinaryOperation1 binary_op1,
                    BinaryOperation2 binary_op2);

  template <class InputIterator, class OutputIterator>
    OutputIterator partial_sum(InputIterator first,
                               InputIterator last,
                               OutputIterator result);
  template <class InputIterator, class OutputIterator,
            class BinaryOperation>
    OutputIterator partial_sum(InputIterator first,
                               InputIterator last,
                               OutputIterator result,
                               BinaryOperation binary_op);

  template <class InputIterator, class OutputIterator>
    OutputIterator adjacent_difference(InputIterator first,
                                       InputIterator last,
                                       OutputIterator result);
  template <class InputIterator, class OutputIterator,
            class BinaryOperation>
    OutputIterator adjacent_difference(InputIterator first,
                                       InputIterator last,
                                       OutputIterator result,
                                       BinaryOperation binary_op);

  template <class ForwardIterator, class T>
    void iota(ForwardIterator first, ForwardIterator last, T value);
}
```

1 The requirements on the types of algorithms' arguments that are described in the introduction to Clause 25 also apply to the following algorithms.

26.144.2 Accumulate [accumulate]

```
template <class InputIterator, class T>
  T accumulate(InputIterator first, InputIterator last, T init);
template <class InputIterator, class T, class BinaryOperation>
  T accumulate(InputIterator first, InputIterator last, T init,
               BinaryOperation binary_op);
```

1

Effects: Computes its result by initializing the accumulator `acc` with the initial value `init` and then

modifies it with `acc = acc + *i` or `acc = binary_op(acc, *i)` for every iterator `i` in the range `[first,last)` in order.[288]

1

Requires: `T` shall meet the requirements of `CopyConstructible` (Table 21) and `CopyAssignable` (Table 23) types. In the range `[first,last]`, `binary_op` shall neither modify elements nor invalidate iterators or subranges.[289]

26.147.1 Inner product [inner.product]

```
template <class InputIterator1, class InputIterator2, class T>
  T inner_product(InputIterator1 first1, InputIterator1 last1,
                  InputIterator2 first2, T init);
template <class InputIterator1, class InputIterator2, class T,
    class BinaryOperation1, class BinaryOperation2>
  T inner_product(InputIterator1 first1, InputIterator1 last1,
                  InputIterator2 first2, T init,
                  BinaryOperation1 binary_op1,
                  BinaryOperation2 binary_op2);
```

1 *Effects:* Computes its result by initializing the accumulator `acc` with the initial value `init` and then modifying it with `acc = acc + (*i1) * (*i2)` or `acc = binary_op1(acc, binary_op2(*i1, *i2))` for every iterator `i1` in the range `[first1,last1)` and iterator `i2` in the range `[first2,first2 + (last1 - first1))` in order.

2 *Requires:* `T` shall meet the requirements of `CopyConstructible` (Table 21) and `CopyAssignable` (Table 23) types. In the ranges `[first1,last1]` and `[first2,first2 + (last1 - first1)]` `binary_op1` and `binary_op2` shall neither modify elements nor invalidate iterators or subranges.[290]

26.147.2 Partial sum [partial.sum]

```
template <class InputIterator, class OutputIterator>
  OutputIterator partial_sum(
    InputIterator first, InputIterator last,
    OutputIterator result);
template
  <class InputIterator, class OutputIterator, class BinaryOperation>
    OutputIterator partial_sum(
      InputIterator first, InputIterator last,
      OutputIterator result, BinaryOperation binary_op);
```

1

Effects: For a non-empty range, the function creates an accumulator `acc` whose type is `InputIterator`'s value type, initializes it with `*first`, and assigns the result to `*result`. For every iterator `i` in `[first + 1,last)` in order, `acc` is then modified by `acc = acc + *i` or `acc = binary_op(acc, *i)` and the result is assigned to `*(result + (i - first))`.

1 *Returns:* `result + (last - first)`.

2 *Complexity:* Exactly `(last - first) - 1` applications of the binary operation.

3 *Requires:* `InputIterator`'s value type shall be constructible from the type of `*first`. The result of the expression `acc + *i` or `binary_op(acc, *i)` shall be implicitly convertible to `InputIterator`'s

288) `accumulate` is similar to the APL reduction operator and Common Lisp reduce function, but it avoids the difficulty of defining the result of reduction on an empty sequence by always requiring an initial value.
289) The use of fully closed ranges is intentional
290) The use of fully closed ranges is intentional

value type. `acc` shall be writable to the `result` output iterator. In the ranges `[first,last]` and `[result,result + (last - first)]` `binary_op` shall neither modify elements nor invalidate iterators or subranges.[291]

4 *Remarks:* `result` may be equal to `first`.

26.148.1 Adjacent difference [adjacent.difference]

```
template <class InputIterator, class OutputIterator>
  OutputIterator adjacent_difference(
    InputIterator first, InputIterator last,
    OutputIterator result);
template <class InputIterator, class OutputIterator, class BinaryOperation>
  OutputIterator adjacent_difference(
    InputIterator first, InputIterator last,
    OutputIterator result,
    BinaryOperation binary_op);
```

1 *Effects:* For a non-empty range, the function creates an accumulator `acc` whose type is `InputIterator`'s value type, initializes it with `*first`, and assigns the result to `*result`. For every iterator `i` in `[first + 1,last)` in order, creates an object `val` whose type is `InputIterator`'s value type, initializes it with `*i`, computes `val - acc` or `binary_op(val, acc)`, assigns the result to `*(result + (i - first))`, and move assigns from `val` to `acc`.

2 *Requires:* `InputIterator`'s value type shall be `MoveAssignable` (Table 22) and shall be constructible from the type of `*first`. `acc` shall be writable to the `result` output iterator. The result of the expression `val - acc` or `binary_op(val, acc)` shall be writable to the `result` output iterator. In the ranges `[first,last]` and `[result,result + (last - first)]`, `binary_op` shall neither modify elements nor invalidate iterators or subranges.[292]

3 *Remarks:* `result` may be equal to `first`.

4 *Returns:* `result + (last - first)`.

5 *Complexity:* Exactly `(last - first) - 1` applications of the binary operation.

26.148.2 Iota [numeric.iota]

```
template <class ForwardIterator, class T>
  void iota(ForwardIterator first, ForwardIterator last, T value);
```

1 *Requires:* `T` shall be convertible to `ForwardIterator`'s value type. The expression `++val`, where `val` has type `T`, shall be well formed.

2 *Effects:* For each element referred to by the iterator `i` in the range `[first,last)`, assigns `*i = value` and increments `value` as if by `++value`.

3 *Complexity:* Exactly `last - first` increments and assignments.

26.149 C library [c.math]

1 The header `<ctgmath>` simply includes the headers `<ccomplex>` and `<cmath>`.

2 [*Note:* The overloads provided in C by type-generic macros are already provided in `<ccomplex>` and `<cmath>` by "sufficient" additional overloads. — *end note*]

3 Tables 119 and 120 describe headers `<cmath>` and `<cstdlib>`, respectively.

4

291) The use of fully closed ranges is intentional.
292) The use of fully closed ranges is intentional.

Table 119 — Header `<cmath>` synopsis

Type	Name(s)			
Macros:				
FP_FAST_FMA	FP_ILOGBNAN	FP_SUBNORMAL	HUGE_VALL	MATH_ERRNO
FP_FAST_FMAF	FP_INFINITE	FP_ZERO	INFINITY	MATH_ERREXCEPT
FP_FAST_FMAL	FP_NAN	HUGE_VAL	NAN	math_errhandling
FP_ILOGB0	FP_NORMAL	HUGE_VALF		
Types:	double_t	float_t		
Math Functions:				
abs	cosh	fmod	logb	remquo
acos	erf	frexp	lrint	rint
acosh	erfc	hypot	lround	round
asin	exp2	ilogb	modf	scalbln
asinh	exp	ldexp	nan	scalbn
atan	expm1	lgamma	nanf	sin
atan2	fabs	llrint	nanl	sinh
atanh	fdim	llround	nearbyint	sqrt
cbrt	floor	log	nextafter	tan
ceil	fma	log10	nexttoward	tanh
copysign	fmax	log1p	pow	tgamma
cos	fmin	log2	remainder	trunc
Classification/comparison Functions:				
fpclassify	isgreaterequal	islessequal	isnan	isunordered
isfinite	isinf	islessgreater	isnormal	signbit
isgreater	isless			

Table 120 — Header `<cstdlib>` synopsis

Type	Name(s)	
Macro:	RAND_MAX	
Types:		
div_t	ldiv_t	lldiv_t
Functions:		
abs	ldiv	rand
div	llabs	srand
labs	lldiv	

The contents of these headers are the same as the Standard C library headers `<math.h>` and `<stdlib.h>` respectively, with the following changes:

1

The `rand` function has the semantics specified in the C standard, except that the implementation may specify that particular library functions may call `rand`. It is implementation-defined whether the `rand` function may introduce data races (17.607.0.1). [*Note:* The random number generation (26.45) facilities in this standard are often preferable to `rand`, because `rand`'s underlying algorithm is unspecified. Use of `rand` therefore continues to be nonportable, with unpredictable and oft-questionable quality and performance. — *end note*]

1

In addition to the `int` versions of certain math functions in `<cstdlib>`, C++ adds `long` and `long long` overloaded versions of these functions, with the same semantics.

1 The added signatures are:

```
long abs(long);                     // labs()
long long abs(long long);           // llabs()
ldiv_t div(long, long);             // ldiv()
lldiv_t div(long long, long long);  // lldiv()
```

2

In addition to the `double` versions of the math functions in `<cmath>`, C++ adds `float` and `long double` overloaded versions of these functions, with the same semantics.

1 The added signatures are:

```
float abs(float);
float acos(float);
float acosh(float);
float asin(float);
float asinh(float);
float atan(float);
float atan2(float, float);
float atanh(float);
float cbrt(float);
float ceil(float);
float copysign(float, float);
float cos(float);
float cosh(float);
float erf(float);
float erfc(float);
float exp(float);
float exp2(float);
float expm1(float);
float fabs(float);
float fdim(float, float);
float floor(float);
float fma(float, float, float);
float fmax(float, float);
float fmin(float, float);
float fmod(float, float);
float frexp(float, int*);
float hypot(float, float);
int ilogb(float);
```

```
float ldexp(float, int);
float lgamma(float);
long long llrint(float);
long long llround(float);
float log(float);
float log10(float);
float log1p(float);
float log2(float);
float logb(float);
long lrint(float);
long lround(float);
float modf(float, float*);
float nearbyint(float);
float nextafter(float, float);
float nexttoward(float, long double);
float pow(float, float);
float remainder(float, float);
float remquo(float, float, int *);
float rint(float);
float round(float);
float scalbln(float, long);
float scalbn(float, int);
float sin(float);
float sinh(float);
float sqrt(float);
float tan(float);
float tanh(float);
float tgamma(float);
float trunc(float);

double abs(double);             // fabs()

long double abs(long double);
long double acos(long double);
long double acosh(long double);
long double asin(long double);
long double asinh(long double);
long double atan(long double);
long double atan2(long double, long double);
long double atanh(long double);
long double cbrt(long double);
long double ceil(long double);
long double copysign(long double, long double);
long double cos(long double);
long double cosh(long double);
long double erf(long double);
long double erfc(long double);
long double exp(long double);
long double exp2(long double);
long double expm1(long double);
long double fabs(long double);
long double fdim(long double, long double);
long double floor(long double);
long double fma(long double, long double, long double);
long double fmax(long double, long double);
```

```
long double fmin(long double, long double);
long double fmod(long double, long double);
long double frexp(long double, int*);
long double hypot(long double, long double);
int ilogb(long double);
long double ldexp(long double, int);
long double lgamma(long double);
long long llrint(long double);
long long llround(long double);
long double log(long double);
long double log10(long double);
long double log1p(long double);
long double log2(long double);
long double logb(long double);
long lrint(long double);
long lround(long double);
long double modf(long double, long double*);
long double nearbyint(long double);
long double nextafter(long double, long double);
long double nexttoward(long double, long double);
long double pow(long double, long double);
long double remainder(long double, long double);
long double remquo(long double, long double, int *);
long double rint(long double);
long double round(long double);
long double scalbln(long double, long);
long double scalbn(long double, int);
long double sin(long double);
long double sinh(long double);
long double sqrt(long double);
long double tan(long double);
long double tanh(long double);
long double tgamma(long double);
long double trunc(long double);
```

2

The classification/comparison functions behave the same as the C macros with the corresponding names defined in 7.12.3, Classification macros, and 7.12.14, Comparison macros in the C Standard. Each function is overloaded for the three floating-point types, as follows:

```
int fpclassify(float x);
bool isfinite(float x);
bool isinf(float x);
bool isnan(float x);
bool isnormal(float x);
bool signbit(float x);

bool isgreater(float x, float y);
bool isgreaterequal(float x, float y);
bool isless(float x, float y);
bool islessequal(float x, float y);
bool islessgreater(float x, float y);
bool isunordered(float x, float y);

int fpclassify(double x);
```

```
bool isfinite(double x);
bool isinf(double x);
bool isnan(double x);
bool isnormal(double x);
bool signbit(double x);

bool isgreater(double x, double y);
bool isgreaterequal(double x, double y);
bool isless(double x, double y);
bool islessequal(double x, double y);
bool islessgreater(double x, double y);
bool isunordered(double x, double y);

int fpclassify(long double x);
bool isfinite(long double x);
bool isinf(long double x);
bool isnan(long double x);
bool isnormal(long double x);
bool signbit(long double x);

bool isgreater(long double x, long double y);
bool isgreaterequal(long double x, long double y);
bool isless(long double x, long double y);
bool islessequal(long double x, long double y);
bool islessgreater(long double x, long double y);
bool isunordered(long double x, long double y);
```

1

Moreover, there shall be additional overloads sufficient to ensure:

1. If any arithmetic argument corresponding to a `double` parameter has type `long double`, then all arithmetic arguments corresponding to `double` parameters are effectively cast to `long double`.
2. Otherwise, if any arithmetic argument corresponding to a `double` parameter has type `double` or an integer type, then all arithmetic arguments corresponding to `double` parameters are effectively cast to `double`.
3. Otherwise, all arithmetic arguments corresponding to `double` parameters have type `float`.

See also: ISO C 7.5, 7.10.2, 7.10.6.

27 Input/output library [input.output]

27.1 General [input.output.general]

1 This Clause describes components that C++ programs may use to perform input/output operations.

2 The following subclauses describe requirements for stream parameters, and components for forward declarations of iostreams, predefined iostreams objects, base iostreams classes, stream buffering, stream formatting and manipulators, string streams, and file streams, as summarized in Table 121.

Table 121 — Input/output library summary

Subclause		Header(s)
27.2	Requirements	
27.16	Forward declarations	`<iosfwd>`
27.17	Standard iostream objects	`<iostream>`
27.56	Iostreams base classes	`<ios>`
27.92	Stream buffers	`<streambuf>`

27.109	Formatting and manipulators	`<istream>`
		`<ostream>`
		`<iomanip>`

27.243	String streams	`<sstream>`

27.248	File streams	`<fstream>`
		`<cstdio>`
		`<cinttypes>`

3 Figure 7 illustrates relationships among various types described in this clause. A line from **A** to **B** indicates that **A** is an alias (e.g. a typedef) for **B** or that **A** is defined in terms of **B**.

Figure 7 — Stream position, offset, and size types [non-normative]

27.2 Iostreams requirements [iostreams.requirements]

27.2.1 Imbue limitations [iostream.limits.imbue]

1 No function described in Clause 27 except for `ios_base::imbue` and `basic_filebuf::pubimbue` causes any instance of `basic_ios::imbue` or `basic_streambuf::imbue` to be called. If any user function called from a function declared in Clause 27 or as an overriding virtual function of any class declared in Clause 27 calls `imbue`, the behavior is undefined.

27.2.2 Positioning type limitations [iostreams.limits.pos]

1 The classes of Clause 27 with template arguments `charT` and `traits` behave as described if `traits::pos_type` and `traits::off_type` are `streampos` and `streamoff` respectively. Except as noted explicitly below, their behavior when `traits::pos_type` and `traits::off_type` are other types is implementation-defined.

2

In the classes of Clause 27, a template parameter with name `charT` represents a member of the set of types containing `char`, `wchar_t`, and any other implementation-defined character types that satisfy the requirements for a character on which any of the iostream components can be instantiated.

27.6.1 Thread safety [iostreams.threadsafety]

1

Concurrent access to a stream object (27.243, 27.248), stream buffer object (27.92), or C Library stream (27.310.1) by multiple threads may result in a data race (1.859) unless otherwise specified (27.17). [*Note:* Data races result in undefined behavior (1.859). — *end note*]

1

If one thread makes a library call a that writes a value to a stream and, as a result, another thread reads this value from the stream through a library call b such that this does not result in a data race, then a's write synchronizes with b's read.

27.16 Forward declarations [iostream.forward]

Header `<iosfwd>` synopsis

```
namespace std {
  template<class charT> class char_traits;
  template<> class char_traits<char>;
  template<> class char_traits<char16_t>;
  template<> class char_traits<char32_t>;
  template<> class char_traits<wchar_t>;

  template<class T> class allocator;

  template <class charT, class traits = char_traits<charT> >
    class basic_ios;
  template <class charT, class traits = char_traits<charT> >
    class basic_streambuf;
  template <class charT, class traits = char_traits<charT> >
    class basic_istream;
  template <class charT, class traits = char_traits<charT> >
    class basic_ostream;
  template <class charT, class traits = char_traits<charT> >
    class basic_iostream;

  template <class charT, class traits = char_traits<charT>,
```

```
    class Allocator = allocator<charT> >
  class basic_stringbuf;
template <class charT, class traits = char_traits<charT>,
    class Allocator = allocator<charT> >
  class basic_istringstream;
template <class charT, class traits = char_traits<charT>,
    class Allocator = allocator<charT> >
  class basic_ostringstream;
template <class charT, class traits = char_traits<charT>,
    class Allocator = allocator<charT> >
  class basic_stringstream;

template <class charT, class traits = char_traits<charT> >
  class basic_filebuf;
template <class charT, class traits = char_traits<charT> >
  class basic_ifstream;
template <class charT, class traits = char_traits<charT> >
  class basic_ofstream;
template <class charT, class traits = char_traits<charT> >
  class basic_fstream;

template <class charT, class traits = char_traits<charT> >
  class istreambuf_iterator;
template <class charT, class traits = char_traits<charT> >
  class ostreambuf_iterator;

typedef basic_ios<char>       ios;
typedef basic_ios<wchar_t>    wios;

typedef basic_streambuf<char> streambuf;
typedef basic_istream<char>   istream;
typedef basic_ostream<char>   ostream;
typedef basic_iostream<char>  iostream;

typedef basic_stringbuf<char>     stringbuf;
typedef basic_istringstream<char> istringstream;
typedef basic_ostringstream<char> ostringstream;
typedef basic_stringstream<char>  stringstream;

typedef basic_filebuf<char>  filebuf;
typedef basic_ifstream<char> ifstream;
typedef basic_ofstream<char> ofstream;
typedef basic_fstream<char>  fstream;

typedef basic_streambuf<wchar_t> wstreambuf;
typedef basic_istream<wchar_t>   wistream;
typedef basic_ostream<wchar_t>   wostream;
typedef basic_iostream<wchar_t>  wiostream;

typedef basic_stringbuf<wchar_t>     wstringbuf;
typedef basic_istringstream<wchar_t> wistringstream;
typedef basic_ostringstream<wchar_t> wostringstream;
typedef basic_stringstream<wchar_t>  wstringstream;

typedef basic_filebuf<wchar_t>  wfilebuf;
```

```
  typedef basic_ifstream<wchar_t> wifstream;
  typedef basic_ofstream<wchar_t> wofstream;
  typedef basic_fstream<wchar_t>  wfstream;

  template <class state> class fpos;
  typedef fpos<char_traits<char>::state_type>    streampos;
  typedef fpos<char_traits<wchar_t>::state_type> wstreampos;
}
```

1 Default template arguments are described as appearing both in `<iosfwd>` and in the synopsis of other headers but it is well-formed to include both `<iosfwd>` and one or more of the other headers.[293]

2 [*Note:* The class template specialization `basic_ios<charT,traits>` serves as a virtual base class for the class templates `basic_istream`, `basic_ostream`, and class templates derived from them. `basic_iostream` is a class template derived from both `basic_istream<charT,traits>` and `basic_ostream<charT,traits>`.

3 The class template specialization `basic_streambuf<charT,traits>` serves as a base class for class templates `basic_stringbuf` and `basic_filebuf`.

4 The class template specialization `basic_istream<charT,traits>` serves as a base class for class templates `basic_istringstream` and `basic_ifstream`.

5 The class template specialization `basic_ostream<charT,traits>` serves as a base class for class templates `basic_ostringstream` and `basic_ofstream`.

6 The class template specialization `basic_iostream<charT,traits>` serves as a base class for class templates `basic_stringstream` and `basic_fstream`.

7 Other typedefs define instances of class templates specialized for `char` or `wchar_t` types.

8 Specializations of the class template `fpos` are used for specifying file position information.

9 The types `streampos` and `wstreampos` are used for positioning streams specialized on `char` and `wchar_t` respectively.

10 This synopsis suggests a circularity between `streampos` and `char_traits<char>`. An implementation can avoid this circularity by substituting equivalent types. One way to do this might be

```
template<class stateT> class fpos { ... };       // depends on nothing
typedef ... _STATE;              // implementation private declaration of stateT

typedef fpos<_STATE> streampos;

template<> struct char_traits<char> {
  typedef streampos
  pos_type;
}
```

— *end note*]

27.17 Standard iostream objects [iostream.objects]

27.17.1 Overview [iostream.objects.overview]

Header `<iostream>` synopsis

```
#include <ios>
#include <streambuf>
```

293) It is the implementation's responsibility to implement headers so that including `<iosfwd>` and other headers does not violate the rules about multiple occurrences of default arguments.

```
#include <istream>
#include <ostream>

namespace std {
  extern istream cin;
  extern ostream cout;
  extern ostream cerr;
  extern ostream clog;

  extern wistream wcin;
  extern wostream wcout;
  extern wostream wcerr;
  extern wostream wclog;
}
```

1

The header `<iostream>` declares objects that associate objects with the standard C streams provided for by the functions declared in `<cstdio>` (27.310.1), and includes all the headers necessary to use these objects.

1

The objects are constructed and the associations are established at some time prior to or during the first time an object of class `ios_base::Init` is constructed, and in any case before the body of `main` begins execution.[294] The objects are not destroyed during program execution.[295] The results of including `<iostream>` in a translation unit shall be as if `<iostream>` defined an instance of `ios_base::Init` with static storage duration. Similarly, the entire program shall behave as if there were at least one instance of `ios_base::Init` with static storage duration.

1

Mixing operations on corresponding wide- and narrow-character streams follows the same semantics as mixing such operations on `FILE`s, as specified in Amendment 1 of the ISO C standard.

1

Concurrent access to a synchronized (27.76.0.1) standard iostream object's formatted and unformatted input (27.109.2.1) and output (27.187.1.1) functions or a standard C stream by multiple threads shall not result in a data race (1.859). [*Note:* Users must still synchronize concurrent use of these objects and streams by multiple threads if they wish to avoid interleaved characters. — *end note*]

27.48.1 Narrow stream objects [narrow.stream.objects]

`istream cin;`

1

The object `cin` controls input from a stream buffer associated with the object `stdin`, declared in `<cstdio>`.

1

After the object `cin` is initialized, `cin.tie()` returns `&cout`. Its state is otherwise the same as required for `basic_ios<char>::init` (27.78.2.2).

`ostream cout;`

294) If it is possible for them to do so, implementations are encouraged to initialize the objects earlier than required.
295) Constructors and destructors for static objects can access these objects to read input from `stdin` or write output to `stdout` or `stderr`.

1 The object `cout` controls output to a stream buffer associated with the object `stdout`, declared in `<cstdio>` (27.310.1).

`ostream cerr;`

1 The object `cerr` controls output to a stream buffer associated with the object `stderr`, declared in `<cstdio>` (27.310.1).

2 After the object `cerr` is initialized, `cerr.flags() & unitbuf` is nonzero and `cerr.tie()` returns `&cout`. Its state is otherwise the same as required for `basic_ios<char>::init` (27.78.2.2).

`ostream clog;`

3 The object `clog` controls output to a stream buffer associated with the object `stderr`, declared in `<cstdio>` (27.310.1).

27.53.1 Wide stream objects [wide.stream.objects]

`wistream wcin;`

1 The object `wcin` controls input from a stream buffer associated with the object `stdin`, declared in `<cstdio>`.

2 After the object `wcin` is initialized, `wcin.tie()` returns `&wcout`. Its state is otherwise the same as required for `basic_ios<wchar_t>::init` (27.78.2.2).

`wostream wcout;`

1 The object `wcout` controls output to a stream buffer associated with the object `stdout`, declared in `<cstdio>` (27.310.1).

`wostream wcerr;`

1 The object `wcerr` controls output to a stream buffer associated with the object `stderr`, declared in `<cstdio>` (27.310.1).

2 After the object `wcerr` is initialized, `wcerr.flags() & unitbuf` is nonzero and `wcerr.tie()` returns `&wcout`. Its state is otherwise the same as required for `basic_ios<wchar_t>::init` (27.78.2.2).

`wostream wclog;`

3 The object `wclog` controls output to a stream buffer associated with the object `stderr`, declared in `<cstdio>` (27.310.1).

27.56 Iostreams base classes [iostreams.base]

27.56.1 Overview [iostreams.base.overview]

Header `<ios>` synopsis

```
#include <iosfwd>

namespace std {
  typedef implementation-defined streamoff;
  typedef implementation-defined streamsize;
  template <class stateT> class fpos;
```

```
  class ios_base;
  template <class charT, class traits = char_traits<charT> >
    class basic_ios;

  // 27.91.1, manipulators:
  ios_base& boolalpha  (ios_base& str);
  ios_base& noboolalpha(ios_base& str);

  ios_base& showbase   (ios_base& str);
  ios_base& noshowbase (ios_base& str);

  ios_base& showpoint  (ios_base& str);
  ios_base& noshowpoint(ios_base& str);

  ios_base& showpos    (ios_base& str);
  ios_base& noshowpos  (ios_base& str);

  ios_base& skipws     (ios_base& str);
  ios_base& noskipws   (ios_base& str);

  ios_base& uppercase  (ios_base& str);
  ios_base& nouppercase(ios_base& str);

  ios_base& unitbuf    (ios_base& str);
  ios_base& nounitbuf  (ios_base& str);

  // 27.91.1.2 adjustfield:
  ios_base& internal   (ios_base& str);
  ios_base& left       (ios_base& str);
  ios_base& right      (ios_base& str);

  // 27.91.1.3 basefield:
  ios_base& dec        (ios_base& str);
  ios_base& hex        (ios_base& str);
  ios_base& oct        (ios_base& str);

  // 27.91.1.4 floatfield:
  ios_base& fixed      (ios_base& str);
  ios_base& scientific (ios_base& str);
  ios_base& hexfloat   (ios_base& str);
  ios_base& defaultfloat(ios_base& str);

  // 27.91.1.5 error reporting:
  enum class io_errc {
    stream = 1
  };

  template <> struct is_error_code_enum<io_errc> : public true_type { };
  error_code make_error_code(io_errc e) noexcept;
  error_condition make_error_condition(io_errc e) noexcept;
  const error_category& iostream_category() noexcept;
}
```

27.56.2 Types [stream.types]

```
typedef implementation-defined streamoff;
```

1 The type `streamoff` is a synonym for one of the signed basic integral types of sufficient size to represent the maximum possible file size for the operating system.[296]

```
typedef implementation-defined streamsize;
```

2 The type `streamsize` is a synonym for one of the signed basic integral types. It is used to represent the number of characters transferred in an I/O operation, or the size of I/O buffers.[297]

27.58.1 Class ios_base [ios.base]

```
namespace std {
  class ios_base {
  public:
    class failure;

    // 27.61.0.0.1 fmtflags
    typedef T1 fmtflags;
    static constexpr fmtflags boolalpha = unspecified ;
    static constexpr fmtflags dec = unspecified ;
    static constexpr fmtflags fixed = unspecified ;
    static constexpr fmtflags hex = unspecified ;
    static constexpr fmtflags internal = unspecified ;
    static constexpr fmtflags left = unspecified ;
    static constexpr fmtflags oct = unspecified ;
    static constexpr fmtflags right = unspecified ;
    static constexpr fmtflags scientific = unspecified ;
    static constexpr fmtflags showbase = unspecified ;
    static constexpr fmtflags showpoint = unspecified ;
    static constexpr fmtflags showpos = unspecified ;
    static constexpr fmtflags skipws = unspecified ;
    static constexpr fmtflags unitbuf = unspecified ;
    static constexpr fmtflags uppercase = unspecified ;
    static constexpr fmtflags adjustfield = see below;
    static constexpr fmtflags basefield = see below;
    static constexpr fmtflags floatfield = see below;

    // 27.61.0.0.2 iostate
    typedef T2 iostate;
    static constexpr iostate badbit = unspecified ;
    static constexpr iostate eofbit = unspecified ;
    static constexpr iostate failbit = unspecified ;
    static constexpr iostate goodbit = see below;

    // 27.66.0.0.1 openmode
    typedef T3 openmode;
    static constexpr openmode app = unspecified ;
    static constexpr openmode ate = unspecified ;
    static constexpr openmode binary = unspecified ;
    static constexpr openmode in = unspecified ;
```

296) Typically `long long`.

297) `streamsize` is used in most places where ISO C would use `size_t`. Most of the uses of `streamsize` could use `size_t`, except for the `strstreambuf` constructors, which require negative values. It should probably be the signed type corresponding to `size_t` (which is what Posix.2 calls `ssize_t`).

```
    static constexpr openmode out = unspecified ;
    static constexpr openmode trunc = unspecified ;

    // 27.69.0.0.1 seekdir
    typedef T4 seekdir;
    static constexpr seekdir beg = unspecified ;
    static constexpr seekdir cur = unspecified ;
    static constexpr seekdir end = unspecified ;

    class Init;

    // 27.72.0.1 fmtflags state:
    fmtflags flags() const;
    fmtflags flags(fmtflags fmtfl);
    fmtflags setf(fmtflags fmtfl);
    fmtflags setf(fmtflags fmtfl, fmtflags mask);
    void unsetf(fmtflags mask);

    streamsize precision() const;
    streamsize precision(streamsize prec);
    streamsize width() const;
    streamsize width(streamsize wide);

    // 27.74.0.1 locales:
    locale imbue(const locale& loc);
    locale getloc() const;

    // 27.77.0.1 storage:
    static int xalloc();
    long&  iword(int index);
    void*& pword(int index);

    // destructor
    virtual ~ios_base();

    // 27.78.0.1 callbacks;
    enum event { erase_event, imbue_event, copyfmt_event };
    typedef void (*event_callback)(event, ios_base&, int index);
    void register_callback(event_callback fn, int index);

    ios_base(const ios_base&) = delete;
    ios_base& operator=(const ios_base&) = delete;

    static bool sync_with_stdio(bool sync = true);

  protected:
    ios_base();

  private:
    static int index;  // exposition only
    long* iarray;      // exposition only
    void** parray;     // exposition only
  };
}
```

1 `ios_base` defines several member types:

(1.1) — a class `failure` derived from `system_error`;

(1.2) — a class `Init`;

(1.3) — three bitmask types, `fmtflags`, `iostate`, and `openmode`;

(1.4) — an enumerated type, `seekdir`.

2

It maintains several kinds of data:

— state information that reflects the integrity of the stream buffer;

— control information that influences how to interpret (format) input sequences and how to generate (format) output sequences;

— additional information that is stored by the program for its private use.

1 [*Note:* For the sake of exposition, the maintained data is presented here as:

(1.1) — `static int index`, specifies the next available unique index for the integer or pointer arrays maintained for the private use of the program, initialized to an unspecified value;

(1.2) — `long* iarray`, points to the first element of an arbitrary-length `long` array maintained for the private use of the program;

(1.3) — `void** parray`, points to the first element of an arbitrary-length pointer array maintained for the private use of the program. — *end note*]

27.59.0.1 Types [ios.types]

27.59.0.1.1 Class `ios_base::failure` [ios::failure]

```
namespace std {
  class ios_base::failure : public system_error {
  public:
    explicit failure(const string& msg, const error_code& ec = io_errc::stream);
    explicit failure(const char* msg, const error_code& ec = io_errc::stream);
  };
}
```

1

The class `failure` defines the base class for the types of all objects thrown as exceptions, by functions in the iostreams library, to report errors detected during stream buffer operations.

1 When throwing `ios_base::failure` exceptions, implementations should provide values of `ec` that identify the specific reason for the failure. [*Note:* Errors arising from the operating system would typically be reported as `system_category()` errors with an error value of the error number reported by the operating system. Errors arising from within the stream library would typically be reported as `error_code(io_errc::stream, iostream_category())`. — *end note*]

`explicit failure(const string& msg, const error_code& ec = io_errc::stream);`

2 *Effects:* Constructs an object of class `failure` by constructing the base class with `msg` and `ec`.

`explicit failure(const char* msg, const error_code& ec = io_errc::stream);`

3 *Effects:* Constructs an object of class `failure` by constructing the base class with `msg` and `ec`.

27.61.0.0.1 Type `ios_base::fmtflags` [ios::fmtflags]

```
typedef T1 fmtflags;
```

1 The type `fmtflags` is a bitmask type (17.20.0.0.1). Setting its elements has the effects indicated in Table 122.

Table 122 — `fmtflags` effects

Element	Effect(s) if set
`boolalpha`	insert and extract `bool` type in alphabetic format
`dec`	converts integer input or generates integer output in decimal base
`fixed`	generate floating-point output in fixed-point notation
`hex`	converts integer input or generates integer output in hexadecimal base
`internal`	adds fill characters at a designated internal point in certain generated output, or identical to `right` if no such point is designated
`left`	adds fill characters on the right (final positions) of certain generated output
`oct`	converts integer input or generates integer output in octal base
`right`	adds fill characters on the left (initial positions) of certain generated output
`scientific`	generates floating-point output in scientific notation
`showbase`	generates a prefix indicating the numeric base of generated integer output
`showpoint`	generates a decimal-point character unconditionally in generated floating-point output
`showpos`	generates a `+` sign in non-negative generated numeric output
`skipws`	skips leading whitespace before certain input operations
`unitbuf`	flushes output after each output operation
`uppercase`	replaces certain lowercase letters with their uppercase equivalents in generated output

2 Type `fmtflags` also defines the constants indicated in Table 123.

Table 123 — `fmtflags` constants

Constant	Allowable values
`adjustfield`	`left | right | internal`
`basefield`	`dec | oct | hex`
`floatfield`	`scientific | fixed`

27.61.0.0.2 Type `ios_base::iostate` [ios::iostate]

```
typedef T2 iostate;
```

1

The type `iostate` is a bitmask type (17.20.0.0.1) that contains the elements indicated in Table 124.

1 Type `iostate` also defines the constant:

(1.1) — `goodbit`, the value zero.

27.66.0.0.1 Type `ios_base::openmode` [ios::openmode]

```
typedef T3 openmode;
```

1

The type `openmode` is a bitmask type (17.20.0.0.1). It contains the elements indicated in Table 125.

Table 124 — `iostate` effects

Element	Effect(s) if set
`badbit`	indicates a loss of integrity in an input or output sequence (such as an irrecoverable read error from a file);
`eofbit`	indicates that an input operation reached the end of an input sequence;
`failbit`	indicates that an input operation failed to read the expected characters, or that an output operation failed to generate the desired characters.

Table 125 — `openmode` effects

Element	Effect(s) if set
`app`	seek to end before each write
`ate`	open and seek to end immediately after opening
`binary`	perform input and output in binary mode (as opposed to text mode)
`in`	open for input
`out`	open for output
`trunc`	truncate an existing stream when opening

27.69.0.0.1 Type `ios_base::seekdir` [ios::seekdir]

```
typedef T4 seekdir;
```

1

The type `seekdir` is an enumerated type (17.17.1.1.2) that contains the elements indicated in Table 126.

Table 126 — `seekdir` effects

Element	Meaning
`beg`	request a seek (for subsequent input or output) relative to the beginning of the stream
`cur`	request a seek relative to the current position within the sequence
`end`	request a seek relative to the current end of the sequence

27.70.0.0.1 Class `ios_base::Init` [ios::Init]

```
namespace std {
  class ios_base::Init {
  public:
    Init();
   ~Init();
  private:
    static int init_cnt; // exposition only
  };
}
```

1 The class `Init` describes an object whose construction ensures the construction of the eight objects declared in `<iostream>` (27.17) that associate file stream buffers with the standard C streams provided for by the functions declared in `<cstdio>` (27.310.1).

2

For the sake of exposition, the maintained data is presented here as:

— `static int init_cnt`, counts the number of constructor and destructor calls for class `Init`, initialized to zero.

`Init();`

1 *Effects:* Constructs an object of class `Init`. Constructs and initializes the objects `cin`, `cout`, `cerr`, `clog`, `wcin`, `wcout`, `wcerr`, and `wclog` if they have not already been constructed and initialized.

`~Init();`

2 *Effects:* Destroys an object of class `Init`. If there are no other instances of the class still in existence, calls `cout.flush()`, `cerr.flush()`, `clog.flush()`, `wcout.flush()`, `wcerr.flush()`, `wclog.flush()`.

27.72.0.1 `ios_base` state functions [fmtflags.state]

`fmtflags flags() const;`

1 *Returns:* The format control information for both input and output.

`fmtflags flags(fmtflags fmtfl);`

2 *Postcondition:* `fmtfl == flags()`.

3 *Returns:* The previous value of `flags()`.

`fmtflags setf(fmtflags fmtfl);`

4 *Effects:* Sets `fmtfl` in `flags()`.

5 *Returns:* The previous value of `flags()`.

`fmtflags setf(fmtflags fmtfl, fmtflags mask);`

6 *Effects:* Clears `mask` in `flags()`, sets `fmtfl & mask` in `flags()`.

1 *Returns:* The previous value of `flags()`.

`void unsetf(fmtflags mask);`

2 *Effects:* Clears `mask` in `flags()`.

`streamsize precision() const;`

3 *Returns:* The precision to generate on certain output conversions.

`streamsize precision(streamsize prec);`

4 *Postcondition:* `prec == precision()`.

5 *Returns:* The previous value of `precision()`.

`streamsize width() const;`

6 *Returns:* The minimum field width (number of characters) to generate on certain output conversions.

`streamsize width(streamsize wide);`

1 *Postcondition:* `wide == width()`.

2 *Returns:* The previous value of `width()`.

27.74.0.1 `ios_base` functions [ios.base.locales]

`locale imbue(const locale& loc);`

1 *Effects:* Calls each registered callback pair `(fn,index)` (27.78.0.1) as `(*fn)(imbue_event,*this,index)` at such a time that a call to `ios_base::getloc()` from within `fn` returns the new locale value `loc`.

2 *Returns:* The previous value of `getloc()`.

3 *Postcondition:* `loc == getloc()`.

`locale getloc() const;`

4

Returns: If no locale has been imbued, a copy of the global C++ locale, `locale()`, in effect at the time of construction. Otherwise, returns the imbued locale, to be used to perform locale-dependent input and output operations.

27.76.0.1 `ios_base` static members [ios.members.static]

`bool sync_with_stdio(bool sync = true);`

1 *Returns:* `true` if the previous state of the standard iostream objects (27.17) was synchronized and otherwise returns `false`. The first time it is called, the function returns `true`.

2

Effects: If any input or output operation has occurred using the standard streams prior to the call, the effect is implementation-defined. Otherwise, called with a false argument, it allows the standard streams to operate independently of the standard C streams.

1 When a standard iostream object `str` is *synchronized* with a standard stdio stream `f`, the effect of inserting a character `c` by

```
fputc(f, c);
```

is the same as the effect of

```
str.rdbuf()->sputc(c);
```

for any sequences of characters; the effect of extracting a character `c` by

```
c = fgetc(f);
```

is the same as the effect of

```
c = str.rdbuf()->sbumpc();
```

for any sequences of characters; and the effect of pushing back a character `c` by

```
ungetc(c, f);
```

is the same as the effect of

```
str.rdbuf()->sputbackc(c);
```

for any sequence of characters.[298]

298) This implies that operations on a standard iostream object can be mixed arbitrarily with operations on the corresponding stdio stream. In practical terms, synchronization usually means that a standard iostream object and a standard stdio object share a buffer.

27.77.0.1 `ios_base` storage functions [ios.base.storage]

```
static int xalloc();
```

1 *Returns:* `index ++`.

2 *Remarks:* Concurrent access to this function by multiple threads shall not result in a data race (1.859).

```
long& iword(int idx);
```

3

Effects: If `iarray` is a null pointer, allocates an array of `long` of unspecified size and stores a pointer to its first element in `iarray`. The function then extends the array pointed at by `iarray` as necessary to include the element `iarray[idx]`. Each newly allocated element of the array is initialized to zero. The reference returned is invalid after any other operations on the object.[299] However, the value of the storage referred to is retained, so that until the next call to `copyfmt`, calling `iword` with the same index yields another reference to the same value. If the function fails[300] and `*this` is a base subobject of a `basic_ios<>` object or subobject, the effect is equivalent to calling `basic_ios<>::setstate(badbit)` on the derived object (which may throw `failure`).

1 *Returns:* On success `iarray[idx]`. On failure, a valid `long&` initialized to 0.

```
void*& pword(int idx);
```

2 *Effects:* If `parray` is a null pointer, allocates an array of pointers to `void` of unspecified size and stores a pointer to its first element in `parray`. The function then extends the array pointed at by `parray` as necessary to include the element `parray[idx]`. Each newly allocated element of the array is initialized to a null pointer. The reference returned is invalid after any other operations on the object. However, the value of the storage referred to is retained, so that until the next call to `copyfmt`, calling `pword` with the same index yields another reference to the same value. If the function fails[301] and `*this` is a base subobject of a `basic_ios<>` object or subobject, the effect is equivalent to calling `basic_ios<>::setstate(badbit)` on the derived object (which may throw `failure`).

3 *Returns:* On success `parray[idx]`. On failure a valid `void*&` initialized to 0.

4 *Remarks:* After a subsequent call to `pword(int)` for the same object, the earlier return value may no longer be valid.

27.78.0.1 `ios_base` callbacks [ios.base.callback]

```
void register_callback(event_callback fn, int index);
```

1 *Effects:* Registers the pair `(fn,index)` such that during calls to `imbue()` (27.74.0.1), `copyfmt()`, or `~ios_base()` (27.78.0.2), the function `fn` is called with argument `index`. Functions registered are called when an event occurs, in opposite order of registration. Functions registered while a callback function is active are not called until the next event.

2 *Requires:* The function `fn` shall not throw exceptions.

Remarks: Identical pairs are not merged. A function registered twice will be called twice.

27.78.0.2 `ios_base` constructors/destructor [ios.base.cons]

```
ios_base();
```

299) An implementation is free to implement both the integer array pointed at by `iarray` and the pointer array pointed at by `parray` as sparse data structures, possibly with a one-element cache for each.
300) for example, because it cannot allocate space.
301) for example, because it cannot allocate space.

1 *Effects:* Each `ios_base` member has an indeterminate value after construction. The object's members shall be initialized by calling `basic_ios::init` before the object's first use or before it is destroyed, whichever comes first; otherwise the behavior is undefined.

`~ios_base();`

2 *Effects:* Destroys an object of class `ios_base`. Calls each registered callback pair `(fn, index)` (27.78.0.1) as `(*fn)(erase_event, *this, index)` at such time that any `ios_base` member function called from within `fn` has well defined results.

27.78.1 Class template `fpos` [fpos]

```
namespace std {
  template <class stateT> class fpos {
  public:
    // 27.78.1.1 Members
    stateT state() const;
    void state(stateT);
  private;
    stateT st; // exposition only
  };
}
```

27.78.1.1 `fpos` members [fpos.members]

`void state(stateT s);`

1 *Effects:* Assign `s` to `st`.

`stateT state() const;`

2 *Returns:* Current value of `st`.

27.78.1.2 `fpos` requirements [fpos.operations]

1 Operations specified in Table 127 are permitted. In that table,

(1.1) — `P` refers to an instance of `fpos`,

(1.2) — `p` and `q` refer to values of type `P`,

(1.3) — `O` refers to type `streamoff`,

(1.4) — `o` refers to a value of type `streamoff`,

(1.5) — `sz` refers to a value of type `streamsize` and

(1.6) — `i` refers to a value of type `int`.

Table 127 — Position type requirements

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
`P(i)`			`p == P(i)` note: a destructor is assumed.
`P p(i);` `P p = i;`			post: `p == P(i)`.

Table 127 — Position type requirements (continued)

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
`P(o)`	`fpos`	converts from `offset`	
`O(p)`	streamoff	converts to `offset`	`P(O(p)) == p`
`p == q`	convertible to `bool`		`==` is an equivalence relation
`p != q`	convertible to `bool`	`!(p == q)`	
`q = p + o` `p += o`	`fpos`	`+` offset	`q - o == p`
`q = p - o` `p -= o`	`fpos`	`-` offset	`q + o == p`
`o = p - q`	streamoff	distance	`q + o == p`
`streamsize(o)`	`streamsize`	converts	`streamsize(O(sz)) == sz`
`O(sz)`	streamoff	converts	`streamsize(O(sz)) == sz`

2 [*Note:* Every implementation is required to supply overloaded operators on `fpos` objects to satisfy the requirements of 27.78.1.2. It is unspecified whether these operators are members of `fpos`, global operators, or provided in some other way. — *end note*]

3 Stream operations that return a value of type `traits::pos_type` return `P(O(-1))` as an invalid value to signal an error. If this value is used as an argument to any `istream`, `ostream`, or `streambuf` member that accepts a value of type `traits::pos_type` then the behavior of that function is undefined.

27.78.2 Class template `basic_ios` [ios]

27.78.2.1 Overview [ios.overview]

```
namespace std {
  template <class charT, class traits = char_traits<charT> >
  class basic_ios : public ios_base {
  public:

    // types:
    typedef charT                     char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;
    typedef traits                    traits_type;

    explicit operator bool() const;
    bool operator!() const;
    iostate rdstate() const;
    void clear(iostate state = goodbit);
    void setstate(iostate state);
    bool good() const;
    bool eof()  const;
    bool fail() const;
    bool bad()  const;

    iostate exceptions() const;
    void exceptions(iostate except);

    // 27.78.2.2 Constructor/destructor:
```

```
    explicit basic_ios(basic_streambuf<charT,traits>* sb);
    virtual ~basic_ios();

    // 27.83.0.1 Members:
    basic_ostream<charT,traits>* tie() const;
    basic_ostream<charT,traits>* tie(basic_ostream<charT,traits>* tiestr);

    basic_streambuf<charT,traits>* rdbuf() const;
    basic_streambuf<charT,traits>* rdbuf(basic_streambuf<charT,traits>* sb);

    basic_ios& copyfmt(const basic_ios& rhs);

    char_type fill() const;
    char_type fill(char_type ch);

    locale imbue(const locale& loc);

    char     narrow(char_type c, char dfault) const;
    char_type widen(char c) const;

    basic_ios(const basic_ios&) = delete;
    basic_ios& operator=(const basic_ios&) = delete;

  protected:
    basic_ios();
    void init(basic_streambuf<charT,traits>* sb);
    void move(basic_ios& rhs);
    void move(basic_ios&& rhs);
    void swap(basic_ios& rhs) noexcept;
    void set_rdbuf(basic_streambuf<charT, traits>* sb);

  };
}
```

27.78.2.2 `basic_ios` constructors [basic.ios.cons]

`explicit basic_ios(basic_streambuf<charT,traits>* sb);`

1 *Effects:* Constructs an object of class `basic_ios`, assigning initial values to its member objects by calling `init(sb)`.

`basic_ios();`

1 *Effects:* Constructs an object of class `basic_ios` (27.78.0.2) leaving its member objects uninitialized. The object shall be initialized by calling `basic_ios::init` before its first use or before it is destroyed, whichever comes first; otherwise the behavior is undefined.

`~basic_ios();`

1 *Remarks:* The destructor does not destroy `rdbuf()`.

`void init(basic_streambuf<charT,traits>* sb);`

Postconditions: The postconditions of this function are indicated in Table 128.

Table 128 — `basic_ios::init()` effects

Element	Value
`rdbuf()`	`sb`
`tie()`	`0`
`rdstate()`	`goodbit` if `sb` is not a null pointer, otherwise `badbit`.
`exceptions()`	`goodbit`
`flags()`	`skipws | dec`
`width()`	`0`
`precision()`	`6`
`fill()`	`widen(' ');`
`getloc()`	a copy of the value returned by `locale()`
`iarray`	a null pointer
`parray`	a null pointer

27.83.0.1 Member functions [basic.ios.members]

`basic_ostream<charT,traits>* tie() const;`

1 *Returns:* An output sequence that is *tied* to (synchronized with) the sequence controlled by the stream buffer.

`basic_ostream<charT,traits>* tie(basic_ostream<charT,traits>* tiestr);`

2 *Requires:* If `tiestr` is not null, `tiestr` must not be reachable by traversing the linked list of tied stream objects starting from `tiestr->tie()`.

3 *Postcondition:* `tiestr == tie()`.

4 *Returns:* The previous value of `tie()`.

`basic_streambuf<charT,traits>* rdbuf() const;`

5 *Returns:* A pointer to the `streambuf` associated with the stream.

`basic_streambuf<charT,traits>* rdbuf(basic_streambuf<charT,traits>* sb);`

6 *Postcondition:* `sb == rdbuf()`.

7 *Effects:* Calls `clear()`.

8 *Returns:* The previous value of `rdbuf()`.

`locale imbue(const locale& loc);`

9 *Effects:* Calls `ios_base::imbue(loc)` (27.74.0.1) and if `rdbuf()!=0` then `rdbuf()->pubimbue(loc)` (27.96.0.2.1).

10 *Returns:* The prior value of `ios_base::imbue()`.

`char narrow(char_type c, char dfault) const;`

11 *Returns:* `use_facet< ctype<char_type> >(getloc()).narrow(c,dfault)`

`char_type widen(char c) const;`

12 *Returns:* `use_facet< ctype<char_type> >(getloc()).widen(c)`

`char_type fill() const;`

13 *Returns:* The character used to pad (fill) an output conversion to the specified field width.

```
char_type fill(char_type fillch);
```

14 *Postcondition:* `traits::eq(fillch, fill())`

15 *Returns:* The previous value of `fill()`.

```
basic_ios& copyfmt(const basic_ios& rhs);
```

16 *Effects:* If `(this == &rhs)` does nothing. Otherwise assigns to the member objects of `*this` the corresponding member objects of `rhs` as follows:

1. calls each registered callback pair `(fn, index)` as `(*fn)(erase_event, *this, index)`;
2. assigns to the member objects of `*this` the corresponding member objects of `rhs`, except that
   - (16.1) — `rdstate()`, `rdbuf()`, and `exceptions()` are left unchanged;
   - (16.2) — the contents of arrays pointed at by `pword` and `iword` are copied, not the pointers themselves;[302] and
   - (16.3) — if any newly stored pointer values in `*this` point at objects stored outside the object `rhs` and those objects are destroyed when `rhs` is destroyed, the newly stored pointer values are altered to point at newly constructed copies of the objects;
3. calls each callback pair that was copied from `rhs` as `(*fn)(copyfmt_event, *this, index)`;
4. calls `exceptions(rhs.except())`.

17 *Note:* The second pass through the callback pairs permits a copied `pword` value to be zeroed, or to have its referent deep copied or reference counted, or to have other special action taken.

18 *Postconditions:* The postconditions of this function are indicated in Table 129.

Table 129 — `basic_ios::copyfmt()` effects

Element	Value
`rdbuf()`	*unchanged*
`tie()`	`rhs.tie()`
`rdstate()`	*unchanged*
`exceptions()`	`rhs.exceptions()`
`flags()`	`rhs.flags()`
`width()`	`rhs.width()`
`precision()`	`rhs.precision()`
`fill()`	`rhs.fill()`
`getloc()`	`rhs.getloc()`

19 *Returns:* `*this`.

```
void move(basic_ios& rhs);
void move(basic_ios&& rhs);
```

20 *Postconditions:* `*this` shall have the state that `rhs` had before the function call, except that `rdbuf()` shall return 0. `rhs` shall be in a valid but unspecified state, except that `rhs.rdbuf()` shall return the same value as it returned before the function call, and `rhs.tie()` shall return 0.

```
void swap(basic_ios& rhs) noexcept;
```

302) This suggests an infinite amount of copying, but the implementation can keep track of the maximum element of the arrays that is non-zero.

21

Effects: The states of `*this` and `rhs` shall be exchanged, except that `rdbuf()` shall return the same value as it returned before the function call, and `rhs.rdbuf()` shall return the same value as it returned before the function call.

```
void set_rdbuf(basic_streambuf<charT, traits>* sb);
```

1 *Requires:* `sb != nullptr`.

2 *Effects:* Associates the `basic_streambuf` object pointed to by `sb` with this stream without calling `clear()`.

3 *Postconditions:* `rdbuf() == sb`.

4 *Throws:* Nothing.

27.84.0.1 `basic_ios` flags functions [iostate.flags]

```
explicit operator bool() const;
```

1 *Returns:* `!fail()`.

```
bool operator!() const;
```

2 *Returns:* `fail()`.

```
iostate rdstate() const;
```

3

Returns: The error state of the stream buffer.

```
void clear(iostate state = goodbit);
```

1

Postcondition: If `rdbuf()!=0` then `state == rdstate()`; otherwise `rdstate()==(state | ios_base::badbit)`.

1

Effects: If `((state | (rdbuf() ? goodbit : badbit)) & exceptions()) == 0`, returns. Otherwise, the function throws an object `fail` of class `basic_ios::failure` (27.59.0.1.1), constructed with implementation-defined argument values.

```
void setstate(iostate state);
```

1

Effects: Calls `clear(rdstate() | state)` (which may throw `basic_ios::failure` (27.59.0.1.1)).

```
bool good() const;
```

1 *Returns:* `rdstate() == 0`

```
bool eof() const;
```

2 *Returns:* `true` if `eofbit` is set in `rdstate()`.

```
bool fail() const;
```

3 *Returns:* `true` if `failbit` or `badbit` is set in `rdstate()`.[303]

```
bool bad() const;
```

1 *Returns:* `true` if `badbit` is set in `rdstate()`.

```
iostate exceptions() const;
```

2 *Returns:* A mask that determines what elements set in `rdstate()` cause exceptions to be thrown.

```
void exceptions(iostate except);
```

1 *Postcondition:* `except == exceptions()`.

2 *Effects:* Calls `clear(rdstate())`.

27.91.1 `ios_base` manipulators [std.ios.manip]

27.91.1.1 `fmtflags` manipulators [fmtflags.manip]

```
ios_base& boolalpha(ios_base& str);
```

1 *Effects:* Calls `str.setf(ios_base::boolalpha)`.

2 *Returns:* `str`.

```
ios_base& noboolalpha(ios_base& str);
```

3 *Effects:* Calls `str.unsetf(ios_base::boolalpha)`.

4 *Returns:* `str`.

```
ios_base& showbase(ios_base& str);
```

5 *Effects:* Calls `str.setf(ios_base::showbase)`.

6 *Returns:* `str`.

```
ios_base& noshowbase(ios_base& str);
```

7 *Effects:* Calls `str.unsetf(ios_base::showbase)`.

8 *Returns:* `str`.

```
ios_base& showpoint(ios_base& str);
```

9 *Effects:* Calls `str.setf(ios_base::showpoint)`.

10 *Returns:* `str`.

```
ios_base& noshowpoint(ios_base& str);
```

11 *Effects:* Calls `str.unsetf(ios_base::showpoint)`.

12 *Returns:* `str`.

```
ios_base& showpos(ios_base& str);
```

13 *Effects:* Calls `str.setf(ios_base::showpos)`.

14 *Returns:* `str`.

303) Checking `badbit` also for `fail()` is historical practice.

`ios_base& noshowpos(ios_base& str);`

15 *Effects:* Calls `str.unsetf(ios_base::showpos)`.

16 *Returns:* `str`.

`ios_base& skipws(ios_base& str);`

17 *Effects:* Calls `str.setf(ios_base::skipws)`.

18 *Returns:* `str`.

`ios_base& noskipws(ios_base& str);`

19 *Effects:* Calls `str.unsetf(ios_base::skipws)`.

20 *Returns:* `str`.

`ios_base& uppercase(ios_base& str);`

21 *Effects:* Calls `str.setf(ios_base::uppercase)`.

22 *Returns:* `str`.

`ios_base& nouppercase(ios_base& str);`

23 *Effects:* Calls `str.unsetf(ios_base::uppercase)`.

24 *Returns:* `str`.

`ios_base& unitbuf(ios_base& str);`

25 *Effects:* Calls `str.setf(ios_base::unitbuf)`.

26 *Returns:* `str`.

`ios_base& nounitbuf(ios_base& str);`

27 *Effects:* Calls `str.unsetf(ios_base::unitbuf)`.

28 *Returns:* `str`.

27.91.1.2 `adjustfield` manipulators [adjustfield.manip]

`ios_base& internal(ios_base& str);`

1 *Effects:* Calls `str.setf(ios_base::internal, ios_base::adjustfield)`.

2 *Returns:* `str`.

`ios_base& left(ios_base& str);`

3 *Effects:* Calls `str.setf(ios_base::left, ios_base::adjustfield)`.

4 *Returns:* `str`.

`ios_base& right(ios_base& str);`

5 *Effects:* Calls `str.setf(ios_base::right, ios_base::adjustfield)`.

6 *Returns:* `str`.

27.91.1.3 `basefield` manipulators [basefield.manip]

`ios_base& dec(ios_base& str);`

1 *Effects:* Calls `str.setf(ios_base::dec, ios_base::basefield)`.

2 *Returns:* `str`[304].

`ios_base& hex(ios_base& str);`

3 *Effects:* Calls `str.setf(ios_base::hex, ios_base::basefield)`.

4 *Returns:* `str`.

`ios_base& oct(ios_base& str);`

5 *Effects:* Calls `str.setf(ios_base::oct, ios_base::basefield)`.

6 *Returns:* `str`.

27.91.1.4 `floatfield` manipulators [floatfield.manip]

`ios_base& fixed(ios_base& str);`

1 *Effects:* Calls `str.setf(ios_base::fixed, ios_base::floatfield)`.

2 *Returns:* `str`.

`ios_base& scientific(ios_base& str);`

3 *Effects:* Calls `str.setf(ios_base::scientific, ios_base::floatfield)`.

4 *Returns:* `str`.

`ios_base& hexfloat(ios_base& str);`

5 *Effects:* Calls `str.setf(ios_base::fixed | ios_base::scientific, ios_base::floatfield)`.

6 *Returns:* `str`.

7 [*Note:* The more obvious use of `ios_base::hex` to specify hexadecimal floating-point format would change the meaning of existing well defined programs. C++2003 gives no meaning to the combination of `fixed` and `scientific`. — *end note*]

`ios_base& defaultfloat(ios_base& str);`

8 *Effects:* Calls `str.unsetf(ios_base::floatfield)`.

9 *Returns:* `str`.

27.91.1.5 Error reporting [error.reporting]

`error_code make_error_code(io_errc e) noexcept;`

1 *Returns:* `error_code(static_cast<int>(e), iostream_category())`.

`error_condition make_error_condition(io_errc e) noexcept;`

2 *Returns:* `error_condition(static_cast<int>(e), iostream_category())`.

`const error_category& iostream_category() noexcept;`

304) The function signature `dec(ios_base&)` can be called by the function signature `basic_ostream& stream::operator<<(ios_base& (*)(ios_base&))` to permit expressions of the form `cout <<dec` to change the format flags stored in `cout`.

3 *Returns:* A reference to an object of a type derived from class `error_category`.

4 The object's `default_error_condition` and `equivalent` virtual functions shall behave as specified for the class `error_category`. The object's `name` virtual function shall return a pointer to the string `"iostream"`.

27.92 Stream buffers [stream.buffers]

27.92.1 Overview [stream.buffers.overview]

Header `<streambuf>` synopsis

```
namespace std {
  template <class charT, class traits = char_traits<charT> >
    class basic_streambuf;
  typedef basic_streambuf<char>     streambuf;
  typedef basic_streambuf<wchar_t> wstreambuf;
}
```

1 The header `<streambuf>` defines types that control input from and output to *character* sequences.

27.92.2 Stream buffer requirements [streambuf.reqts]

1

Stream buffers can impose various constraints on the sequences they control. Some constraints are:

— The controlled input sequence can be not readable.

— The controlled output sequence can be not writable.

— The controlled sequences can be associated with the contents of other representations for character sequences, such as external files.

— The controlled sequences can support operations *directly* to or from associated sequences.

— The controlled sequences can impose limitations on how the program can read characters from a sequence, write characters to a sequence, put characters back into an input sequence, or alter the stream position.

1

Each sequence is characterized by three pointers which, if non-null, all point into the same `charT` array object. The array object represents, at any moment, a (sub)sequence of characters from the sequence. Operations performed on a sequence alter the values stored in these pointers, perform reads and writes directly to or from associated sequences, and alter "the stream position" and conversion state as needed to maintain this subsequence relationship. The three pointers are:

— the *beginning pointer*, or lowest element address in the array (called `xbeg` here);

— the *next pointer*, or next element address that is a current candidate for reading or writing (called `xnext` here);

— the *end pointer*, or first element address beyond the end of the array (called `xend` here).

1 The following semantic constraints shall always apply for any set of three pointers for a sequence, using the pointer names given immediately above:

(1.1) — If `xnext` is not a null pointer, then `xbeg` and `xend` shall also be non-null pointers into the same `charT` array, as described above; otherwise, `xbeg` and `xend` shall also be null.

(1.2) — If `xnext` is not a null pointer and `xnext < xend` for an output sequence, then a *write position* is available. In this case, `*xnext` shall be assignable as the next element to write (to put, or to store a character value, into the sequence).

(1.3) — If `xnext` is not a null pointer and `xbeg < xnext` for an input sequence, then a *putback position* is available. In this case, `xnext[-1]` shall have a defined value and is the next (preceding) element to store a character that is put back into the input sequence.

(1.4) — If `xnext` is not a null pointer and `xnext < xend` for an input sequence, then a *read position* is available. In this case, `*xnext` shall have a defined value and is the next element to read (to get, or to obtain a character value, from the sequence).

27.95.1 Class template `basic_streambuf<charT,traits>` [streambuf]

```
namespace std {
  template <class charT, class traits = char_traits<charT> >
  class basic_streambuf {
  public:

    // types:
    typedef charT                     char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;
    typedef traits                    traits_type;

    virtual ~basic_streambuf();

    // 27.96.0.2.1 locales:
    locale   pubimbue(const locale& loc);
    locale   getloc() const;

    // 27.96.0.2.2 buffer and positioning:
    basic_streambuf<char_type,traits>*
       pubsetbuf(char_type* s, streamsize n);
    pos_type pubseekoff(off_type off, ios_base::seekdir way,
      ios_base::openmode which =
          ios_base::in | ios_base::out);
    pos_type pubseekpos(pos_type sp,
      ios_base::openmode which =
          ios_base::in | ios_base::out);
    int      pubsync();

    // Get and put areas:
    // 27.96.0.2.3 Get area:
    streamsize in_avail();
    int_type snextc();
    int_type sbumpc();
    int_type sgetc();
    streamsize sgetn(char_type* s, streamsize n);

    // 27.96.0.2.4 Putback:
    int_type sputbackc(char_type c);
```

```
    int_type sungetc();

    // 27.97.0.0.1 Put area:
    int_type   sputc(char_type c);
    streamsize sputn(const char_type* s, streamsize n);

  protected:
    basic_streambuf();
    basic_streambuf(const basic_streambuf& rhs);
    basic_streambuf& operator=(const basic_streambuf& rhs);

    void swap(basic_streambuf& rhs);

    // 27.99.0.1.2 Get area:
    char_type* eback() const;
    char_type* gptr()  const;
    char_type* egptr() const;
    void       gbump(int n);
    void       setg(char_type* gbeg, char_type* gnext, char_type* gend);

    // 27.99.0.1.3 Put area:
    char_type* pbase() const;
    char_type* pptr() const;
    char_type* epptr() const;
    void       pbump(int n);
    void       setp(char_type* pbeg, char_type* pend);

    // 27.99.0.2 virtual functions:
    // 27.99.0.2.1 Locales:
    virtual void imbue(const locale& loc);

    // 27.99.0.2.2 Buffer management and positioning:
    virtual basic_streambuf<char_type,traits>*
         setbuf(char_type* s, streamsize n);
    virtual pos_type seekoff(off_type off, ios_base::seekdir way,
        ios_base::openmode which = ios_base::in | ios_base::out);
    virtual pos_type seekpos(pos_type sp,
        ios_base::openmode which = ios_base::in | ios_base::out);
    virtual int      sync();

    // 27.101.0.0.1 Get area:
    virtual streamsize showmanyc();
    virtual streamsize xsgetn(char_type* s, streamsize n);
    virtual int_type   underflow();
    virtual int_type   uflow();

    // 27.102.0.0.1 Putback:
    virtual int_type   pbackfail(int_type c = traits::eof());

    // 27.103.0.0.1 Put area:
    virtual streamsize xsputn(const char_type* s, streamsize n);
    virtual int_type   overflow (int_type c = traits::eof());
  };
}
```

1

The class template `basic_streambuf<charT,traits>` serves as an abstract base class for deriving various *stream buffers* whose objects each control two *character sequences*:

— a character *input sequence*;

— a character *output sequence*.

27.96.0.1 `basic_streambuf` constructors [streambuf.cons]

`basic_streambuf();`

1 *Effects:* Constructs an object of class `basic_streambuf<charT,traits>` and initializes:[305]

(1.1) — all its pointer member objects to null pointers,

(1.2) — the `getloc()` member to a copy the global locale, `locale()`, at the time of construction.

2 *Remarks:* Once the `getloc()` member is initialized, results of calling locale member functions, and of members of facets so obtained, can safely be cached until the next time the member `imbue` is called.

`basic_streambuf(const basic_streambuf& rhs);`

3 *Effects:* Constructs a copy of `rhs`.

4 *Postconditions:*

(4.1) — `eback() == rhs.eback()`

(4.2) — `gptr() == rhs.gptr()`

(4.3) — `egptr() == rhs.egptr()`

(4.4) — `pbase() == rhs.pbase()`

(4.5) — `pptr() == rhs.pptr()`

(4.6) — `epptr() == rhs.epptr()`

(4.7) — `getloc() == rhs.getloc()`

`~basic_streambuf();`

5 *Effects:* None.

27.96.0.2 `basic_streambuf` public member functions [streambuf.members]

27.96.0.2.1 Locales [streambuf.locales]

`locale pubimbue(const locale& loc);`

1 *Postcondition:* `loc == getloc()`.

2 *Effects:* Calls `imbue(loc)`.

3 *Returns:* Previous value of `getloc()`.

`locale getloc() const;`

4 *Returns:* If `pubimbue()` has ever been called, then the last value of `loc` supplied, otherwise the current global locale, `locale()`, in effect at the time of construction. If called after `pubimbue()` has been called but before `pubimbue` has returned (i.e., from within the call of `imbue()`) then it returns the previous value.

305) The default constructor is protected for class `basic_streambuf` to assure that only objects for classes derived from this class may be constructed.

27.96.0.2.2 Buffer management and positioning [streambuf.buffer]

```
basic_streambuf<char_type,traits>* pubsetbuf(char_type* s, streamsize n);
```

1 *Returns:* `setbuf(s, n)`.

```
pos_type pubseekoff(off_type off, ios_base::seekdir way,
               ios_base::openmode which = ios_base::in | ios_base::out);
```

2 *Returns:* `seekoff(off, way, which)`.

```
pos_type pubseekpos(pos_type sp,
               ios_base::openmode which = ios_base::in | ios_base::out);
```

3 *Returns:* `seekpos(sp, which)`.

```
int pubsync();
```

4 *Returns:* `sync()`.

27.96.0.2.3 Get area [streambuf.pub.get]

```
streamsize in_avail();
```

1 *Returns:* If a read position is available, returns `egptr() - gptr()`. Otherwise returns `showmanyc()` (27.101.0.0.1).

```
int_type snextc();
```

2 *Effects:* Calls `sbumpc()`.

3 *Returns:* If that function returns `traits::eof()`, returns `traits::eof()`. Otherwise, returns `sgetc()`.

```
int_type sbumpc();
```

4 *Returns:* If the input sequence read position is not available, returns `uflow()`. Otherwise, returns `traits::to_int_type(*gptr())` and increments the next pointer for the input sequence.

```
int_type sgetc();
```

5 *Returns:* If the input sequence read position is not available, returns `underflow()`. Otherwise, returns `traits::to_int_type(*gptr())`.

```
streamsize sgetn(char_type* s, streamsize n);
```

6 *Returns:* `xsgetn(s, n)`.

27.96.0.2.4 Putback [streambuf.pub.pback]

```
int_type sputbackc(char_type c);
```

1 *Returns:* If the input sequence putback position is not available, or if `traits::eq(c,gptr()[-1])` is false, returns `pbackfail(traits::to_int_type(c))`. Otherwise, decrements the next pointer for the input sequence and returns `traits::to_int_type(*gptr())`.

```
int_type sungetc();
```

2

Returns: If the input sequence putback position is not available, returns `pbackfail()`. Otherwise, decrements the next pointer for the input sequence and returns `traits::to_int_type(*gptr())`.

27.97.0.0.1 Put area [streambuf.pub.put]

```
int_type sputc(char_type c);
```

1 *Returns:* If the output sequence write position is not available, returns `overflow(traits::to_int_type(c))`. Otherwise, stores `c` at the next pointer for the output sequence, increments the pointer, and returns `traits::to_int_type(c)`.

```
streamsize sputn(const char_type* s, streamsize n);
```

1 *Returns:* `xsputn(s,n)`.

27.99.0.1 basic_streambuf protected member functions [streambuf.protected]

27.99.0.1.1 Assignment [streambuf.assign]

```
basic_streambuf& operator=(const basic_streambuf& rhs);
```

1 *Effects:* Assigns the data members of `rhs` to `*this`.

2 *Postconditions:*

(2.1) — `eback() == rhs.eback()`

(2.2) — `gptr() == rhs.gptr()`

(2.3) — `egptr() == rhs.egptr()`

(2.4) — `pbase() == rhs.pbase()`

(2.5) — `pptr() == rhs.pptr()`

(2.6) — `epptr() == rhs.epptr()`

(2.7) — `getloc() == rhs.getloc()`

3 *Returns:* `*this`.

```
void swap(basic_streambuf& rhs);
```

4 *Effects:* Swaps the data members of `rhs` and `*this`.

27.99.0.1.2 Get area access [streambuf.get.area]

```
char_type* eback() const;
```

1 *Returns:* The beginning pointer for the input sequence.

```
char_type* gptr() const;
```

2 *Returns:* The next pointer for the input sequence.

```
char_type* egptr() const;
```

3 *Returns:* The end pointer for the input sequence.

```
void gbump(int n);
```

4 *Effects:* Adds `n` to the next pointer for the input sequence.

```
void setg(char_type* gbeg, char_type* gnext, char_type* gend);
```

5 *Postconditions:* `gbeg == eback()`, `gnext == gptr()`, and `gend == egptr()`.

27.99.0.1.3 Put area access [streambuf.put.area]

```
char_type* pbase() const;
```

1 *Returns:* The beginning pointer for the output sequence.

```
char_type* pptr() const;
```

2 *Returns:* The next pointer for the output sequence.

```
char_type* epptr() const;
```

3 *Returns:* The end pointer for the output sequence.

```
void pbump(int n);
```

4 *Effects:* Adds `n` to the next pointer for the output sequence.

```
void setp(char_type* pbeg, char_type* pend);
```

5 *Postconditions:* `pbeg == pbase()`, `pbeg == pptr()`, and `pend == epptr()`.

27.99.0.2 `basic_streambuf` virtual functions [streambuf.virtuals]

27.99.0.2.1 Locales [streambuf.virt.locales]

```
void imbue(const locale&);
```

1 *Effects:* Change any translations based on locale.

2 *Remarks:* Allows the derived class to be informed of changes in locale at the time they occur. Between invocations of this function a class derived from streambuf can safely cache results of calls to locale functions and to members of facets so obtained.

3 *Default behavior:* Does nothing.

27.99.0.2.2 Buffer management and positioning [streambuf.virt.buffer]

```
basic_streambuf* setbuf(char_type* s, streamsize n);
```

1

Effects: Influences stream buffering in a way that is defined separately for each class derived from `basic_streambuf` in this Clause (27.246.0.1, 27.288.0.1).

1 *Default behavior:* Does nothing. Returns `this`.

```
pos_type seekoff(off_type off, ios_base::seekdir way,
                 ios_base::openmode which
                  = ios_base::in | ios_base::out);
```

2 *Effects:* Alters the stream positions within one or more of the controlled sequences in a way that is defined separately for each class derived from `basic_streambuf` in this Clause (27.246.0.1, 27.288.0.1).

3 *Default behavior:* Returns `pos_type(off_type(-1))`.

```
pos_type seekpos(pos_type sp,
                 ios_base::openmode which
                  = ios_base::in | ios_base::out);
```

4 *Effects:* Alters the stream positions within one or more of the controlled sequences in a way that is defined separately for each class derived from `basic_streambuf` in this Clause (27.243.2, 27.248.1.1).

5 *Default behavior:* Returns `pos_type(off_type(-1))`.

```
int sync();
```

6 *Effects:* Synchronizes the controlled sequences with the arrays. That is, if `pbase()` is non-null the characters between `pbase()` and `pptr()` are written to the controlled sequence. The pointers may then be reset as appropriate.

7 *Returns:*-1 on failure. What constitutes failure is determined by each derived class (27.288.0.1).

8 *Default behavior:* Returns zero.

27.101.0.0.1 Get area [streambuf.virt.get]

```
streamsize showmanyc();
```[306]

1

Returns: An estimate of the number of characters available in the sequence, or -1. If it returns a positive value, then successive calls to `underflow()` will not return `traits::eof()` until at least that number of characters have been extracted from the stream. If `showmanyc()` returns -1, then calls to `underflow()` or `uflow()` will fail.[307]

1 *Default behavior:* Returns zero.

2 *Remarks:* Uses `traits::eof()`.

```
streamsize xsgetn(char_type* s, streamsize n);
```

3 *Effects:* Assigns up to `n` characters to successive elements of the array whose first element is designated by `s`. The characters assigned are read from the input sequence as if by repeated calls to `sbumpc()`. Assigning stops when either `n` characters have been assigned or a call to `sbumpc()` would return `traits::eof()`.

4 *Returns:* The number of characters assigned.[308]

5 *Remarks:* Uses `traits::eof()`.

```
int_type underflow();
```

6 *Remarks:* The public members of `basic_streambuf` call this virtual function only if `gptr()` is null or `gptr() >= egptr()`

7 *Returns:* `traits::to_int_type(c)`, where `c` is the first *character* of the *pending sequence*, without moving the input sequence position past it. If the pending sequence is null then the function returns `traits::eof()` to indicate failure.

8 The *pending sequence* of characters is defined as the concatenation of:

a) If `gptr()` is non-null, then the `egptr() - gptr()` characters starting at `gptr()`, otherwise the empty sequence.

b) Some sequence (possibly empty) of characters read from the input sequence.

9 The *result character* is

a) If the pending sequence is non-empty, the first character of the sequence.

b) If the pending sequence is empty then the next character that would be read from the input sequence.

306) The morphemes of `showmanyc` are "es-how-many-see", not "show-manic".
307) `underflow` or `uflow` might fail by throwing an exception prematurely. The intention is not only that the calls will not return `eof()` but that they will return "immediately."
308) Classes derived from `basic_streambuf` can provide more efficient ways to implement `xsgetn()` and `xsputn()` by overriding these definitions from the base class.

10 The *backup sequence* is defined as the concatenation of:

a) If `eback()` is null then empty,
b) Otherwise the `gptr() - eback()` characters beginning at `eback()`.

11 *Effects:* The function sets up the `gptr()` and `egptr()` satisfying one of:

a) If the pending sequence is non-empty, `egptr()` is non-null and `egptr() - gptr()` characters starting at `gptr()` are the characters in the pending sequence
b) If the pending sequence is empty, either `gptr()` is null or `gptr()` and `egptr()` are set to the same non-null pointer value.

12 If `eback()` and `gptr()` are non-null then the function is not constrained as to their contents, but the "usual backup condition" is that either:

a) If the backup sequence contains at least `gptr() - eback()` characters, then the `gptr() - eback()` characters starting at `eback()` agree with the last `gptr() - eback()` characters of the backup sequence.
b) Or the `n` characters starting at `gptr() - n` agree with the backup sequence (where `n` is the length of the backup sequence)

13 *Default behavior:* Returns `traits::eof()`.

```
int_type uflow();
```

14 *Requires:* The constraints are the same as for `underflow()`, except that the result character shall be transferred from the pending sequence to the backup sequence, and the pending sequence shall not be empty before the transfer.

15 *Default behavior:* Calls `underflow()`. If `underflow()` returns `traits::eof()`, returns `traits::eof()`. Otherwise, returns the value of `traits::to_int_type(*gptr())` and increment the value of the next pointer for the input sequence.

16 *Returns:* `traits::eof()` to indicate failure.

27.102.0.0.1 Putback [streambuf.virt.pback]

```
int_type pbackfail(int_type c = traits::eof());
```

1

Remarks: The public functions of `basic_streambuf` call this virtual function only when `gptr()` is null, `gptr() == eback()`, or `traits::eq(traits::to_char_type(c),gptr()[-1])` returns `false`. Other calls shall also satisfy that constraint.

The *pending sequence* is defined as for `underflow()`, with the modifications that

— If `traits::eq_int_type(c,traits::eof())` returns `true`, then the input sequence is backed up one character before the pending sequence is determined.
— If `traits::eq_int_type(c,traits::eof())` returns `false`, then `c` is prepended. Whether the input sequence is backed up or modified in any other way is unspecified.

1 *Postcondition:* On return, the constraints of `gptr()`, `eback()`, and `pptr()` are the same as for `underflow()`.

2 *Returns:* `traits::eof()` to indicate failure. Failure may occur because the input sequence could not be backed up, or if for some other reason the pointers could not be set consistent with the constraints. `pbackfail()` is called only when put back has really failed.

3 Returns some value other than `traits::eof()` to indicate success.

4 *Default behavior:* Returns `traits::eof()`.

27.103.0.0.1 Put area [streambuf.virt.put]

```
streamsize xsputn(const char_type* s, streamsize n);
```

1 *Effects:* Writes up to `n` characters to the output sequence as if by repeated calls to `sputc(c)`. The characters written are obtained from successive elements of the array whose first element is designated by `s`. Writing stops when either `n` characters have been written or a call to `sputc(c)` would return `traits::eof()`. Is is unspecified whether the function calls `overflow()` when `pptr() == epptr()` becomes true or whether it achieves the same effects by other means.

2 *Returns:* The number of characters written.

```
int_type overflow(int_type c = traits::eof());
```

3 *Effects:* Consumes some initial subsequence of the characters of the *pending sequence*. The pending sequence is defined as the concatenation of

a) if `pbase()` is null then the empty sequence otherwise, `pptr() - pbase()` characters beginning at `pbase()`.

b) if `traits::eq_int_type(c,traits::eof())` returns `true`, then the empty sequence otherwise, the sequence consisting of `c`.

4 *Remarks:* The member functions `sputc()` and `sputn()` call this function in case that no room can be found in the put buffer enough to accommodate the argument character sequence.

5 *Requires:* Every overriding definition of this virtual function shall obey the following constraints:

1) The effect of consuming a character on the associated output sequence is specified[309]

2) Let `r` be the number of characters in the pending sequence not consumed. If `r` is non-zero then `pbase()` and `pptr()` shall be set so that: `pptr() - pbase() == r` and the `r` characters starting at `pbase()` are the associated output stream. In case `r` is zero (all characters of the pending sequence have been consumed) then either `pbase()` is set to `nullptr`, or `pbase()` and `pptr()` are both set to the same non-null value.

3) The function may fail if either appending some character to the associated output stream fails or if it is unable to establish `pbase()` and `pptr()` according to the above rules.

1 *Returns:* `traits::eof()` or throws an exception if the function fails.

Otherwise, returns some value other than `traits::eof()` to indicate success.[310]

1 *Default behavior:* Returns `traits::eof()`.

27.109 Formatting and manipulators [iostream.format]

27.109.1 Overview [iostream.format.overview]

Header `<istream>` synopsis

309) That is, for each class derived from an instance of `basic_streambuf` in this Clause (27.243.2, 27.248.1.1), a specification of how consuming a character effects the associated output sequence is given. There is no requirement on a program-defined class.

310) Typically, `overflow` returns `c` to indicate success, except when `traits::eq_int_type(c,traits::eof())` returns `true`, in which case it returns `traits::not_eof(c)`.

```
namespace std {
  template <class charT, class traits = char_traits<charT> >
    class basic_istream;
  typedef basic_istream<char>     istream;
  typedef basic_istream<wchar_t> wistream;

  template <class charT, class traits = char_traits<charT> >
    class basic_iostream;
  typedef basic_iostream<char>    iostream;
  typedef basic_iostream<wchar_t> wiostream;

  template <class charT, class traits>
    basic_istream<charT,traits>& ws(basic_istream<charT,traits>& is);

  template <class charT, class traits, class T>
    basic_istream<charT, traits>&
    operator>>(basic_istream<charT, traits>&& is, T& x);
}
```

Header `<ostream>` synopsis

```
namespace std {
  template <class charT, class traits = char_traits<charT> >
    class basic_ostream;
  typedef basic_ostream<char>     ostream;
  typedef basic_ostream<wchar_t> wostream;

  template <class charT, class traits>
    basic_ostream<charT,traits>& endl(basic_ostream<charT,traits>& os);
  template <class charT, class traits>
    basic_ostream<charT,traits>& ends(basic_ostream<charT,traits>& os);
  template <class charT, class traits>
    basic_ostream<charT,traits>& flush(basic_ostream<charT,traits>& os);

  template <class charT, class traits, class T>
    basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>&& os, const T& x);
}
```

Header `<iomanip>` synopsis

```
namespace std {
  // types T1, T2, ... are unspecified implementation types
  T1 resetiosflags(ios_base::fmtflags mask);
  T2 setiosflags  (ios_base::fmtflags mask);
  T3 setbase(int base);
  template<charT> T4 setfill(charT c);
  T5 setprecision(int n);
  T6 setw(int n);
  template <class moneyT> T7 get_money(moneyT& mon, bool intl = false);
  template <class moneyT> T8 put_money(const moneyT& mon, bool intl = false);
  template <class charT> T9 get_time(struct tm* tmb, const charT* fmt);
  template <class charT> T10 put_time(const struct tm* tmb, const charT* fmt);

  template <class charT>
    T11 quoted(const charT* s, charT delim=charT('"'), charT escape=charT('\\'));
```

```
  template <class charT, class traits, class Allocator>
    T12 quoted(const basic_string<charT, traits, Allocator>& s,
               charT delim=charT('"'), charT escape=charT('\\'));

  template <class charT, class traits, class Allocator>
    T13 quoted(basic_string<charT, traits, Allocator>& s,
               charT delim=charT('"'), charT escape=charT('\\'));
}
```

27.109.2 Input streams [input.streams]

1 The header `<istream>` defines two types and a function signature that control input from a stream buffer along with a function template that extracts from stream rvalues.

27.109.2.1 Class template `basic_istream` [istream]

```
namespace std {
  template <class charT, class traits = char_traits<charT> >
  class basic_istream : virtual public basic_ios<charT,traits> {
  public:
    // types (inherited from basic_ios (27.78.2)):
    typedef charT                     char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;
    typedef traits                    traits_type;

    // 27.116.0.0.1 Constructor/destructor:
    explicit basic_istream(basic_streambuf<charT,traits>* sb);
    virtual ~basic_istream();

    // 27.117.0.0.2 Prefix/suffix:
    class sentry;

    // 27.128.0.1 Formatted input:
    basic_istream<charT,traits>& operator>>(
      basic_istream<charT,traits>& (*pf)(basic_istream<charT,traits>&));
    basic_istream<charT,traits>& operator>>(
                       basic_ios<charT,traits>& (*pf)(basic_ios<charT,traits>&));
    basic_istream<charT,traits>& operator>>(
      ios_base& (*pf)(ios_base&));

    basic_istream<charT,traits>& operator>>(bool& n);
    basic_istream<charT,traits>& operator>>(short& n);
    basic_istream<charT,traits>& operator>>(unsigned short& n);
    basic_istream<charT,traits>& operator>>(int& n);
    basic_istream<charT,traits>& operator>>(unsigned int& n);
    basic_istream<charT,traits>& operator>>(long& n);
    basic_istream<charT,traits>& operator>>(unsigned long& n);
    basic_istream<charT,traits>& operator>>(long long& n);
    basic_istream<charT,traits>& operator>>(unsigned long long& n);
    basic_istream<charT,traits>& operator>>(float& f);
    basic_istream<charT,traits>& operator>>(double& f);
    basic_istream<charT,traits>& operator>>(long double& f);
```

```
  basic_istream<charT,traits>& operator>>(void*& p);
  basic_istream<charT,traits>& operator>>(
    basic_streambuf<char_type,traits>* sb);

  // 27.154.0.1 Unformatted input:
  streamsize gcount() const;
  int_type get();
  basic_istream<charT,traits>& get(char_type& c);
  basic_istream<charT,traits>& get(char_type* s, streamsize n);
  basic_istream<charT,traits>& get(char_type* s, streamsize n,
                                   char_type delim);
  basic_istream<charT,traits>& get(basic_streambuf<char_type,traits>& sb);
  basic_istream<charT,traits>& get(basic_streambuf<char_type,traits>& sb,
                                   char_type delim);

  basic_istream<charT,traits>& getline(char_type* s, streamsize n);
  basic_istream<charT,traits>& getline(char_type* s, streamsize n,
                                       char_type delim);

  basic_istream<charT,traits>& ignore(
    streamsize n = 1, int_type delim = traits::eof());
  int_type                     peek();
  basic_istream<charT,traits>& read    (char_type* s, streamsize n);
  streamsize                   readsome(char_type* s, streamsize n);

  basic_istream<charT,traits>& putback(char_type c);
  basic_istream<charT,traits>& unget();
  int sync();

  pos_type tellg();
  basic_istream<charT,traits>& seekg(pos_type);
  basic_istream<charT,traits>& seekg(off_type, ios_base::seekdir);

protected:
  basic_istream(const basic_istream& rhs) = delete;
  basic_istream(basic_istream&& rhs);

  // 27.117.0.0.1 Assign/swap:
  basic_istream& operator=(const basic_istream& rhs) = delete;
  basic_istream& operator=(basic_istream&& rhs);
  void swap(basic_istream& rhs);
};

// 27.141.0.0.1 character extraction templates:
template<class charT, class traits>
  basic_istream<charT,traits>& operator>>(basic_istream<charT,traits>&,
                                          charT&);
template<class traits>
  basic_istream<char,traits>& operator>>(basic_istream<char,traits>&,
                                         unsigned char&);
template<class traits>
  basic_istream<char,traits>& operator>>(basic_istream<char,traits>&,
                                         signed char&);

template<class charT, class traits>
```

```
    basic_istream<charT,traits>& operator>>(basic_istream<charT,traits>&,
                                            charT*);
  template<class traits>
    basic_istream<char,traits>& operator>>(basic_istream<char,traits>&,
                                           unsigned char*);
  template<class traits>
    basic_istream<char,traits>& operator>>(basic_istream<char,traits>&,
                                           signed char*);
}
```

1 The class `basic_istream` defines a number of member function signatures that assist in reading and interpreting input from sequences controlled by a stream buffer.

2

Two groups of member function signatures share common properties: the *formatted input functions* (or *extractors*) and the *unformatted input functions.* Both groups of input functions are described as if they obtain (or *extract*) input *characters* by calling `rdbuf()->sbumpc()` or `rdbuf()->sgetc()`. They may use other public members of `istream`.

1

If `rdbuf()->sbumpc()` or `rdbuf()->sgetc()` returns `traits::eof()`, then the input function, except as explicitly noted otherwise, completes its actions and does `setstate(eofbit)`, which may throw `ios_base::failure` (27.84.0.1), before returning.

1

If one of these called functions throws an exception, then unless explicitly noted otherwise, the input function sets `badbit` in error state. If `badbit` is on in `exceptions()`, the input function rethrows the exception without completing its actions, otherwise it does not throw anything and proceeds as if the called function had returned a failure indication.

27.116.0.0.1 `basic_istream` constructors [istream.cons]

`explicit basic_istream(basic_streambuf<charT,traits>* sb);`

1 *Effects:* Constructs an object of class `basic_istream`, assigning initial values to the base class by calling `basic_ios::init(sb)` (27.78.2.2).

2 *Postcondition:* `gcount() == 0`

`basic_istream(basic_istream&& rhs);`

3 *Effects:* Move constructs from the rvalue `rhs`. This is accomplished by default constructing the base class, copying the `gcount()` from `rhs`, calling `basic_ios<charT, traits>::move(rhs)` to initialize the base class, and setting the `gcount()` for `rhs` to 0.

`virtual ~basic_istream();`

4 *Effects:* Destroys an object of class `basic_istream`.

5

Remarks: Does not perform any operations of `rdbuf()`.

27.117.0.0.1 Class `basic_istream` assign and swap [istream.assign]

`basic_istream& operator=(basic_istream&& rhs);`

1 *Effects:* `swap(rhs);`.

2 *Returns:* `*this`.

```
void swap(basic_istream& rhs);
```

3 *Effects:* Calls `basic_ios<charT, traits>::swap(rhs)`. Exchanges the values returned by `gcount()` and `rhs.gcount()`.

27.117.0.0.2 Class `basic_istream::sentry` [istream::sentry]

```
namespace std {
  template <class charT,class traits = char_traits<charT> >
  class basic_istream<charT,traits>::sentry {
    typedef traits traits_type;
    bool ok_; // exposition only
  public:
    explicit sentry(basic_istream<charT,traits>& is, bool noskipws = false);
    ~sentry();
    explicit operator bool() const { return ok_; }
    sentry(const sentry&) = delete;
    sentry& operator=(const sentry&) = delete;
  };
}
```

1 The class `sentry` defines a class that is responsible for doing exception safe prefix and suffix operations.

```
explicit sentry(basic_istream<charT,traits>& is, bool noskipws = false);
```

1 *Effects:* If `is.good()` is `false`, calls `is.setstate(failbit)`. Otherwise, prepares for formatted or unformatted input. First, if `is.tie()` is not a null pointer, the function calls `is.tie()->flush()` to synchronize the output sequence with any associated external C stream. Except that this call can be suppressed if the put area of `is.tie()` is empty. Further an implementation is allowed to defer the call to `flush` until a call of `is.rdbuf()->underflow()` occurs. If no such call occurs before the `sentry` object is destroyed, the call to `flush` may be eliminated entirely.[311] If `noskipws` is zero and `is.flags() & ios_base::skipws` is nonzero, the function extracts and discards each character as long as the next available input character `c` is a whitespace character. If `is.rdbuf()->sbumpc()` or `is.rdbuf()->sgetc()` returns `traits::eof()`, the function calls `setstate(failbit | eofbit)` (which may throw `ios_base::failure`).

1 *Remarks:* The constructor `explicit sentry(basic_istream<charT,traits>& is, bool noskipws = false)` uses the currently imbued locale in `is`, to determine whether the next input character is whitespace or not.

2 To decide if the character `c` is a whitespace character, the constructor performs as if it executes the following code fragment:

```
const ctype<charT>& ctype = use_facet<ctype<charT> >(is.getloc());
if (ctype.is(ctype.space,c)!=0)
  // c is a whitespace character.
```

3 If, after any preparation is completed, `is.good()` is `true`, `ok_ != false` otherwise, `ok_ == false`. During preparation, the constructor may call `setstate(failbit)` (which may throw `ios_base::failure` (27.84.0.1))[312]

311) This will be possible only in functions that are part of the library. The semantics of the constructor used in user code is as specified.

312) The sentry constructor and destructor can also perform additional implementation-dependent operations.

```
~sentry();
```

4

Effects: None.

```
explicit operator bool() const;
```

1 *Effects:* Returns `ok_`.

27.128.0.1 Formatted input functions [istream.formatted]

27.128.0.1.1 Common requirements [istream.formatted.reqmts]

1

Each formatted input function begins execution by constructing an object of class `sentry` with the `noskipws` (second) argument `false`. If the `sentry` object returns `true`, when converted to a value of type `bool`, the function endeavors to obtain the requested input. If an exception is thrown during input then `ios::badbit` is turned on[313] in `*this`'s error state. If `(exceptions()&badbit) != 0` then the exception is rethrown. In any case, the formatted input function destroys the `sentry` object. If no exception has been thrown, it returns `*this`.

27.132.0.0.1 Arithmetic extractors [istream.formatted.arithmetic]

```
operator>>(unsigned short& val);
operator>>(unsigned int& val);
operator>>(long& val);
operator>>(unsigned long& val);
operator>>(long long& val);
operator>>(unsigned long long& val);
operator>>(float& val);
operator>>(double& val);
operator>>(long double& val);
operator>>(bool& val);
operator>>(void*& val);
```

1

As in the case of the inserters, these extractors depend on the locale's `num_get<>` (22.45.0.1) object to perform parsing the input stream data. These extractors behave as formatted input functions (as described in 27.128.0.1.1). After a sentry object is constructed, the conversion occurs as if performed by the following code fragment:

```
typedef num_get< charT,istreambuf_iterator<charT,traits> > numget;
iostate err = iostate::goodbit;
use_facet< numget >(loc).get(*this, 0, *this, err, val);
setstate(err);
```

In the above fragment, `loc` stands for the private member of the `basic_ios` class. [*Note:* The first argument provides an object of the `istreambuf_iterator` class which is an iterator pointed to an input stream. It bypasses istreams and uses streambufs directly. — *end note*] Class `locale` relies on this type as its interface to `istream`, so that it does not need to depend directly on `istream`.

```
operator>>(short& val);
```

1

The conversion occurs as if performed by the following code fragment (using the same notation as for the preceding code fragment):

313) This is done without causing an `ios::failure` to be thrown.

```
typedef num_get<charT,istreambuf_iterator<charT,traits> > numget;
iostate err = ios_base::goodbit;
long lval;
use_facet<numget>(loc).get(*this, 0, *this, err, lval);
if (lval < numeric_limits<short>::min()) {
  err |= ios_base::failbit;
  val = numeric_limits<short>::min();
} else if (numeric_limits<short>::max() < lval) {
  err |= ios_base::failbit;
  val = numeric_limits<short>::max();
}  else
  val = static_cast<short>(lval);
setstate(err);
```

`operator>>(int& val);`

1 The conversion occurs as if performed by the following code fragment (using the same notation as for the preceding code fragment):

```
typedef num_get<charT,istreambuf_iterator<charT,traits> > numget;
iostate err = ios_base::goodbit;
long lval;
use_facet<numget>(loc).get(*this, 0, *this, err, lval);
if (lval < numeric_limits<int>::min()) {
  err |= ios_base::failbit;
  val = numeric_limits<int>::min();
} else if (numeric_limits<int>::max() < lval) {
  err |= ios_base::failbit;
  val = numeric_limits<int>::max();
}  else
  val = static_cast<int>(lval);
setstate(err);
```

27.141.0.0.1 `basic_istream::operator>>` [istream::extractors]

`basic_istream<charT,traits>& operator>>`
`(basic_istream<charT,traits>& (*pf)(basic_istream<charT,traits>&));`

1 *Effects:* None. This extractor does not behave as a formatted input function (as described in 27.128.0.1.1.)

2 *Returns:* `pf(*this)`.[314]

`basic_istream<charT,traits>& operator>>`
`(basic_ios<charT,traits>& (*pf)(basic_ios<charT,traits>&));`

3 *Effects:* Calls `pf(*this)`. This extractor does not behave as a formatted input function (as described in 27.128.0.1.1).

4 *Returns:* `*this`.

`basic_istream<charT,traits>& operator>>`
`(ios_base& (*pf)(ios_base&));`

5 *Effects:* Calls `pf(*this)`.[315] This extractor does not behave as a formatted input function (as described in 27.128.0.1.1).

6 *Returns:* `*this`.

314) See, for example, the function signature `ws(basic_istream&)` (27.184.0.1).
315) See, for example, the function signature `dec(ios_base&)` (27.91.1.3).

```
template<class charT, class traits>
  basic_istream<charT,traits>& operator>>(basic_istream<charT,traits>& in,
                                          charT* s);
template<class traits>
  basic_istream<char,traits>& operator>>(basic_istream<char,traits>& in,
                                         unsigned char* s);
template<class traits>
  basic_istream<char,traits>& operator>>(basic_istream<char,traits>& in,
                                         signed char* s);
```

7 *Effects:* Behaves like a formatted input member (as described in 27.128.0.1.1) of `in`. After a `sentry` object is constructed, `operator>>` extracts characters and stores them into successive locations of an array whose first element is designated by `s`. If `width()` is greater than zero, `n` is `width()`. Otherwise `n` is the number of elements of the largest array of `char_type` that can store a terminating `charT()`. `n` is the maximum number of characters stored.

8 Characters are extracted and stored until any of the following occurs:

— `n-1` characters are stored;

— end of file occurs on the input sequence;

— `ct.is(ct.space,c)` is `true` for the next available input character `c`, where `ct` is `use_facet<ctype< charT> >(in.getloc())`.

1 `operator>>` then stores a null byte (`charT()`) in the next position, which may be the first position if no characters were extracted. `operator>>` then calls `width(0)`.

1 If the function extracted no characters, it calls `setstate(failbit)`, which may throw `ios_base::failure` (27.84.0.1).

2 *Returns:* `in`.

```
template<class charT, class traits>
  basic_istream<charT,traits>& operator>>(basic_istream<charT,traits>& in,
                                          charT& c);
template<class traits>
  basic_istream<char,traits>& operator>>(basic_istream<char,traits>& in,
                                         unsigned char& c);
template<class traits>
  basic_istream<char,traits>& operator>>(basic_istream<char,traits>& in,
                                         signed char& c);
```

3 *Effects:* Behaves like a formatted input member (as described in 27.128.0.1.1) of `in`. After a `sentry` object is constructed a character is extracted from `in`, if one is available, and stored in `c`. Otherwise, the function calls `in.setstate(failbit)`.

1 *Returns:* `in`.

```
basic_istream<charT,traits>& operator>>
  (basic_streambuf<charT,traits>* sb);
```

2 *Effects:* Behaves as an unformatted input function (as described in 27.154.0.1, paragraph 1). If `sb` is null, calls `setstate(failbit)`, which may throw `ios_base::failure` (27.84.0.1). After a sentry

object is constructed, extracts characters from `*this` and inserts them in the output sequence controlled by `sb`. Characters are extracted and inserted until any of the following occurs:

— end-of-file occurs on the input sequence;

— inserting in the output sequence fails (in which case the character to be inserted is not extracted);

— an exception occurs (in which case the exception is caught).

1

If the function inserts no characters, it calls `setstate(failbit)`, which may throw `ios_base::failure` (27.84.0.1). If it inserted no characters because it caught an exception thrown while extracting characters from `*this` and `failbit` is on in `exceptions()` (27.84.0.1), then the caught exception is rethrown.

1 *Returns:* `*this`.

27.154.0.1 Unformatted input functions [istream.unformatted]

1

Each unformatted input function begins execution by constructing an object of class `sentry` with the default argument `noskipws` (second) argument `true`. If the `sentry` object returns `true`, when converted to a value of type `bool`, the function endeavors to obtain the requested input. Otherwise, if the sentry constructor exits by throwing an exception or if the sentry object returns false, when converted to a value of type `bool`, the function returns without attempting to obtain any input. In either case the number of extracted characters is set to 0; unformatted input functions taking a character array of non-zero size as an argument shall also store a null character (using `charT()`) in the first location of the array. If an exception is thrown during input then `ios::badbit` is turned on[316] in `*this`'s error state. (Exceptions thrown from `basic_ios<>::clear()` are not caught or rethrown.) If `(exceptions()&badbit) != 0` then the exception is rethrown. It also counts the number of characters extracted. If no exception has been thrown it ends by storing the count in a member object and returning the value specified. In any event the `sentry` object is destroyed before leaving the unformatted input function.

```
streamsize gcount() const;
```

1

Effects: None. This member function does not behave as an unformatted input function (as described in 27.154.0.1, paragraph 1).

1 *Returns:* The number of characters extracted by the last unformatted input member function called for the object.

```
int_type get();
```

2

Effects: Behaves as an unformatted input function (as described in 27.154.0.1, paragraph 1). After constructing a sentry object, extracts a character `c`, if one is available. Otherwise, the function calls `setstate(failbit)`, which may throw `ios_base::failure` (27.84.0.1),

1 *Returns:* `c` if available, otherwise `traits::eof()`.

```
basic_istream<charT,traits>& get(char_type& c);
```

316) This is done without causing an `ios::failure` to be thrown.

2

Effects: Behaves as an unformatted input function (as described in 27.154.0.1, paragraph 1). After constructing a sentry object, extracts a character, if one is available, and assigns it to `c`.[317] Otherwise, the function calls `setstate(failbit)` (which may throw `ios_base::failure` (27.84.0.1)).

1 *Returns:* `*this`.

```
basic_istream<charT,traits>& get(char_type* s, streamsize n,
                  char_type delim );
```

2

Effects: Behaves as an unformatted input function (as described in 27.154.0.1, paragraph 1). After constructing a sentry object, extracts characters and stores them into successive locations of an array whose first element is designated by `s`.[318] Characters are extracted and stored until any of the following occurs:

— `n` is less than one or `n - 1` characters are stored;

— end-of-file occurs on the input sequence (in which case the function calls `setstate(eofbit)`);

— `traits::eq(c, delim)` for the next available input character `c` (in which case `c` is not extracted).

1 If the function stores no characters, it calls `setstate(failbit)` (which may throw `ios_base::failure` (27.84.0.1)). In any case, if `n` is greater than zero it then stores a null character into the next successive location of the array.

2 *Returns:* `*this`.

```
basic_istream<charT,traits>& get(char_type* s, streamsize n);
```

3 *Effects:* Calls `get(s,n,widen('\n'))`

4 *Returns:* Value returned by the call.

```
basic_istream<charT,traits>& get(basic_streambuf<char_type,traits>& sb,
                  char_type delim );
```

5 *Effects:* Behaves as an unformatted input function (as described in 27.154.0.1, paragraph 1). After constructing a sentry object, extracts characters and inserts them in the output sequence controlled by `sb`. Characters are extracted and inserted until any of the following occurs:

(5.1) — end-of-file occurs on the input sequence;

(5.2) — inserting in the output sequence fails (in which case the character to be inserted is not extracted);

(5.3) — `traits::eq(c, delim)` for the next available input character `c` (in which case `c` is not extracted);

(5.4) — an exception occurs (in which case, the exception is caught but not rethrown).

6 If the function inserts no characters, it calls `setstate(failbit)`, which may throw `ios_base::failure` (27.84.0.1).

7 *Returns:* `*this`.

```
basic_istream<charT,traits>& get(basic_streambuf<char_type,traits>& sb);
```

8 *Effects:* Calls `get(sb, widen('\n'))`

9 *Returns:* Value returned by the call.

317) Note that this function is not overloaded on types `signed char` and `unsigned char`.
318) Note that this function is not overloaded on types `signed char` and `unsigned char`.

```
basic_istream<charT,traits>& getline(char_type* s, streamsize n,
                      char_type delim);
```

10

Effects: Behaves as an unformatted input function (as described in 27.154.0.1, paragraph 1). After constructing a sentry object, extracts characters and stores them into successive locations of an array whose first element is designated by `s`.[319] Characters are extracted and stored until one of the following occurs:

1. end-of-file occurs on the input sequence (in which case the function calls `setstate(eofbit)`);
2. `traits::eq(c, delim)` for the next available input character `c` (in which case the input character is extracted but not stored);[320]
3. `n` is less than one or `n - 1` characters are stored (in which case the function calls `setstate(failbit)`).

1 These conditions are tested in the order shown.[321]

2 If the function extracts no characters, it calls `setstate(failbit)` (which may throw `ios_base::failure` (27.84.0.1)).[322]

3 In any case, if `n` is greater than zero, it then stores a null character (using `charT()`) into the next successive location of the array.

4 *Returns:* `*this`.

5 [*Example:*

```
#include <iostream>

int main() {
  using namespace std;
  const int line_buffer_size = 100;

  char buffer[line_buffer_size];
  int line_number = 0;
  while (cin.getline(buffer, line_buffer_size, '\n') || cin.gcount()) {
    int count = cin.gcount();
    if (cin.eof())
      cout << "Partial final line";   // cin.fail() is false
    else if (cin.fail()) {
      cout << "Partial long line";
      cin.clear(cin.rdstate() & ~ios_base::failbit);
    } else {
      count--;                        // Don't include newline in count
      cout << "Line " << ++line_number;
    }
    cout << " (" << count << " chars): " << buffer << endl;
  }
}
```

— *end example*]

319) Note that this function is not overloaded on types `signed char` and `unsigned char`.
320) Since the final input character is "extracted," it is counted in the `gcount()`, even though it is not stored.
321) This allows an input line which exactly fills the buffer, without setting `failbit`. This is different behavior than the historical AT&T implementation.
322) This implies an empty input line will not cause `failbit` to be set.

```
basic_istream<charT,traits>& getline(char_type* s, streamsize n);
```

6 *Returns:* `getline(s,n,widen('\n'))`

```
basic_istream<charT,traits>&
    ignore(streamsize n = 1, int_type delim = traits::eof());
```

7

Effects: Behaves as an unformatted input function (as described in 27.154.0.1, paragraph 1). After constructing a sentry object, extracts characters and discards them. Characters are extracted until any of the following occurs:

— `n != numeric_limits<streamsize>::max()` (18.41.2) and `n` characters have been extracted so far

— end-of-file occurs on the input sequence (in which case the function calls `setstate(eofbit)`, which may throw `ios_base::failure` (27.84.0.1));

— `traits::eq_int_type(traits::to_int_type(c), delim)` for the next available input character `c` (in which case `c` is extracted).

1 *Remarks:* The last condition will never occur if `traits::eq_int_type(delim, traits::eof())`.

2 *Returns:* `*this`.

```
int_type peek();
```

3 *Effects:* Behaves as an unformatted input function (as described in 27.154.0.1, paragraph 1). After constructing a sentry object, reads but does not extract the current input character.

4 *Returns:* `traits::eof()` if `good()` is `false`. Otherwise, returns `rdbuf()->sgetc()`.

```
basic_istream<charT,traits>& read(char_type* s, streamsize n);
```

5

Effects: Behaves as an unformatted input function (as described in 27.154.0.1, paragraph 1). After constructing a sentry object, if `!good()` calls `setstate(failbit)` which may throw an exception, and return. Otherwise extracts characters and stores them into successive locations of an array whose first element is designated by `s`.[323] Characters are extracted and stored until either of the following occurs:

— `n` characters are stored;

— end-of-file occurs on the input sequence (in which case the function calls `setstate(failbit | eofbit)`, which may throw `ios_base::failure` (27.84.0.1)).

1 *Returns:* `*this`.

```
streamsize readsome(char_type* s, streamsize n);
```

2

Effects: Behaves as an unformatted input function (as described in 27.154.0.1, paragraph 1). After constructing a sentry object, if `!good()` calls `setstate(failbit)` which may throw an exception, and return. Otherwise extracts characters and stores them into successive locations of an array whose first element is designated by `s`. If `rdbuf()->in_avail() == -1`, calls `setstate(eofbit)` (which may throw `ios_base::failure` (27.84.0.1)), and extracts no characters;

— If `rdbuf()->in_avail() == 0`, extracts no characters

323) Note that this function is not overloaded on types `signed char` and `unsigned char`.

— If `rdbuf()->in_avail() > 0`, extracts `min(rdbuf()->in_avail(),n))`.

1 *Returns:* The number of characters extracted.

`basic_istream<charT,traits>& putback(char_type c);`

2

Effects: Behaves as an unformatted input function (as described in 27.154.0.1, paragraph 1), except that the function first clears `eofbit`. After constructing a sentry object, if `!good()` calls `setstate(failbit)` which may throw an exception, and return. If `rdbuf()` is not null, calls `rdbuf->sputbackc()`. If `rdbuf()` is null, or if `sputbackc()` returns `traits::eof()`, calls `setstate(badbit)` (which may throw `ios_base::failure` (27.84.0.1)). [*Note:* This function extracts no characters, so the value returned by the next call to `gcount()` is 0. — *end note*]

1 *Returns:* `*this`.

`basic_istream<charT,traits>& unget();`

2

Effects: Behaves as an unformatted input function (as described in 27.154.0.1, paragraph 1), except that the function first clears `eofbit`. After constructing a sentry object, if `!good()` calls `setstate(failbit)` which may throw an exception, and return. If `rdbuf()` is not null, calls `rdbuf()->sungetc()`. If `rdbuf()` is null, or if `sungetc()` returns `traits::eof()`, calls `setstate(badbit)` (which may throw `ios_base::failure` (27.84.0.1)). [*Note:* This function extracts no characters, so the value returned by the next call to `gcount()` is 0. — *end note*]

1 *Returns:* `*this`.

`int sync();`

2 *Effects:* Behaves as an unformatted input function (as described in 27.154.0.1, paragraph 1), except that it does not count the number of characters extracted and does not affect the value returned by subsequent calls to `gcount()`. After constructing a sentry object, if `rdbuf()` is a null pointer, returns -1 . Otherwise, calls `rdbuf()->pubsync()` and, if that function returns -1 calls `setstate(badbit)` (which may throw `ios_base::failure` (27.84.0.1), and returns `-1`. Otherwise, returns zero.

`pos_type tellg();`

3 *Effects:* Behaves as an unformatted input function (as described in 27.154.0.1, paragraph 1), except that it does not count the number of characters extracted and does not affect the value returned by subsequent calls to `gcount()`.

4

Returns: After constructing a sentry object, if `fail() != false`, returns `pos_type(-1)` to indicate failure. Otherwise, returns `rdbuf()->pubseekoff(0, cur, in)`.

`basic_istream<charT,traits>& seekg(pos_type pos);`

1

Effects: Behaves as an unformatted input function (as described in 27.154.0.1, paragraph 1), except that the function first clears `eofbit`, it does not count the number of characters extracted, and it does not affect the value returned by subsequent calls to `gcount()`. After constructing a sentry object, if `fail() != true`, executes `rdbuf()->pubseekpos(pos, ios_base::in)`. In case of failure, the function calls `setstate(failbit)` (which may throw `ios_base::failure`).

1 *Returns:* `*this`.

```
basic_istream<charT,traits>& seekg(off_type off, ios_base::seekdir dir);
```

2

Effects: Behaves as an unformatted input function (as described in 27.154.0.1, paragraph 1), except that it does not count the number of characters extracted and does not affect the value returned by subsequent calls to `gcount()`. After constructing a sentry object, if `fail() != true`, executes `rdbuf()->pubseekoff(off, dir, ios_base::in)`. In case of failure, the function calls `setstate(failbit)` (which may throw `ios_base::failure`).

1 *Returns:* `*this`.

27.184.0.1 Standard `basic_istream` manipulators [istream.manip]

```
namespace std {
  template <class charT, class traits>
    basic_istream<charT,traits>& ws(basic_istream<charT,traits>& is);
}
```

1

Effects: Behaves as an unformatted input function (as described in 27.154.0.1, paragraph 1), except that it does not count the number of characters extracted and does not affect the value returned by subsequent calls to is.gcount(). After constructing a sentry object extracts characters as long as the next available character `c` is whitespace or until there are no more characters in the sequence. Whitespace characters are distinguished with the same criterion as used by `sentry::sentry` (27.117.0.0.2). If `ws` stops extracting characters because there are no more available it sets `eofbit`, but not `failbit`.

1 *Returns:* `is`.

27.186.0.1 Class template `basic_iostream` [iostreamclass]

```
namespace std {
  template <class charT, class traits = char_traits<charT> >
  class basic_iostream :
    public basic_istream<charT,traits>,
    public basic_ostream<charT,traits> {
  public:
    // types:
    typedef charT                     char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;
    typedef traits                    traits_type;

    // constructor/destructor
    explicit basic_iostream(basic_streambuf<charT,traits>* sb);
    virtual ~basic_iostream();

  protected:
    basic_iostream(const basic_iostream& rhs) = delete;
    basic_iostream(basic_iostream&& rhs);

    // assign/swap
    basic_iostream& operator=(const basic_iostream& rhs) = delete;
    basic_iostream& operator=(basic_iostream&& rhs);
    void swap(basic_iostream& rhs);
  };
}
```

1 The class `basic_iostream` inherits a number of functions that allow reading input and writing output to sequences controlled by a stream buffer.

27.186.0.1.1 `basic_iostream` constructors [iostream.cons]

```
explicit basic_iostream(basic_streambuf<charT,traits>* sb);
```

1 *Effects:* Constructs an object of class `basic_iostream`, assigning initial values to the base classes by calling `basic_istream<charT,traits>(sb)` (27.109.2.1) and `basic_ostream<charT,traits>(sb)` (27.187.1.1)

2 *Postcondition:* `rdbuf()==sb` and `gcount()==0`.

```
basic_iostream(basic_iostream&& rhs);
```

3

Effects: Move constructs from the rvalue `rhs` by constructing the `basic_istream` base class with `move(rhs)`.

27.187.0.0.1 `basic_iostream` destructor [iostream.dest]

```
virtual ~basic_iostream();
```

1 *Effects:* Destroys an object of class `basic_iostream`.

2 *Remarks:* Does not perform any operations on `rdbuf()`.

27.187.0.0.2 `basic_iostream` assign and swap [iostream.assign]

```
basic_iostream& operator=(basic_iostream&& rhs);
```

1 *Effects:* `swap(rhs)`.

```
void swap(basic_iostream& rhs);
```

2 *Effects:* Calls `basic_istream<charT, traits>::swap(rhs)`.

27.187.0.1 Rvalue stream extraction [istream.rvalue]

```
template <class charT, class traits, class T>
  basic_istream<charT, traits>&
  operator>>(basic_istream<charT, traits>&& is, T& x);
```

1 *Effects:* `is >>x`

2 *Returns:* `is`

27.187.1 Output streams [output.streams]

1 The header `<ostream>` defines a type and several function signatures that control output to a stream buffer along with a function template that inserts into stream rvalues.

27.187.1.1 Class template `basic_ostream` [ostream]

```
namespace std {
  template <class charT, class traits = char_traits<charT> >
  class basic_ostream : virtual public basic_ios<charT,traits> {
  public:
    // types (inherited from basic_ios (27.78.2)):
    typedef charT                     char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
```

```
  typedef typename traits::off_type off_type;
  typedef traits                    traits_type;

  // 27.188.0.1 Constructor/destructor:
  explicit basic_ostream(basic_streambuf<char_type,traits>* sb);
  virtual ~basic_ostream();

  // 27.190.0.2 Prefix/suffix:
  class sentry;

  // 27.198.0.1 Formatted output:
  basic_ostream<charT,traits>& operator<<(
    basic_ostream<charT,traits>& (*pf)(basic_ostream<charT,traits>&));
  basic_ostream<charT,traits>& operator<<(
    basic_ios<charT,traits>& (*pf)(basic_ios<charT,traits>&));
  basic_ostream<charT,traits>& operator<<(
    ios_base& (*pf)(ios_base&));

  basic_ostream<charT,traits>& operator<<(bool n);
  basic_ostream<charT,traits>& operator<<(short n);
  basic_ostream<charT,traits>& operator<<(unsigned short n);
  basic_ostream<charT,traits>& operator<<(int n);
  basic_ostream<charT,traits>& operator<<(unsigned int n);
  basic_ostream<charT,traits>& operator<<(long n);
  basic_ostream<charT,traits>& operator<<(unsigned long n);
  basic_ostream<charT,traits>& operator<<(long long n);
  basic_ostream<charT,traits>& operator<<(unsigned long long n);
  basic_ostream<charT,traits>& operator<<(float f);
  basic_ostream<charT,traits>& operator<<(double f);
  basic_ostream<charT,traits>& operator<<(long double f);

  basic_ostream<charT,traits>& operator<<(const void* p);
  basic_ostream<charT,traits>& operator<<(
    basic_streambuf<char_type,traits>* sb);

  // 27.225.0.1 Unformatted output:
  basic_ostream<charT,traits>& put(char_type c);
  basic_ostream<charT,traits>& write(const char_type* s, streamsize n);

  basic_ostream<charT,traits>& flush();

  // 27.196.0.1 seeks:
  pos_type tellp();
  basic_ostream<charT,traits>& seekp(pos_type);
  basic_ostream<charT,traits>& seekp(off_type, ios_base::seekdir);
protected:
  basic_ostream(const basic_ostream& rhs) = delete;
  basic_ostream(basic_ostream&& rhs);

  // 27.190.0.1 Assign/swap
  basic_ostream& operator=(const basic_ostream& rhs) = delete;
  basic_ostream& operator=(basic_ostream&& rhs);
  void swap(basic_ostream& rhs);
};
```

```
  // 27.217.0.0.1 character inserters
  template<class charT, class traits>
    basic_ostream<charT,traits>& operator<<(basic_ostream<charT,traits>&,
                                            charT);
  template<class charT, class traits>
    basic_ostream<charT,traits>& operator<<(basic_ostream<charT,traits>&,
                                            char);
  template<class traits>
    basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>&,
                                           char);

  // signed and unsigned
  template<class traits>
    basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>&,
                                           signed char);
  template<class traits>
    basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>&,
                                           unsigned char);

  template<class charT, class traits>
    basic_ostream<charT,traits>& operator<<(basic_ostream<charT,traits>&,
                                            const charT*);
  template<class charT, class traits>
    basic_ostream<charT,traits>& operator<<(basic_ostream<charT,traits>&,
                                            const char*);
  template<class traits>
    basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>&,
                                           const char*);

  // signed and unsigned
  template<class traits>
    basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>&,
                                           const signed char*);
  template<class traits>
    basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>&,
                                           const unsigned char*);
}
```

1 The class `basic_ostream` defines a number of member function signatures that assist in formatting and writing output to output sequences controlled by a stream buffer.

2

Two groups of member function signatures share common properties: the *formatted output functions* (or *inserters*) and the *unformatted output functions.* Both groups of output functions generate (or *insert*) output *characters* by actions equivalent to calling `rdbuf()->sputc(int_type)`. They may use other public members of `basic_ostream` except that they shall not invoke any virtual members of `rdbuf()` except `overflow()`, `xsputn()`, and `sync()`.

1 If one of these called functions throws an exception, then unless explicitly noted otherwise the output function sets `badbit` in error state. If `badbit` is on in `exceptions()`, the output function rethrows the exception without completing its actions, otherwise it does not throw anything and treat as an error.

27.188.0.1 `basic_ostream` constructors [ostream.cons]

```
explicit basic_ostream(basic_streambuf<charT,traits>* sb);
```

1

Effects: Constructs an object of class `basic_ostream`, assigning initial values to the base class by calling `basic_ios<charT,traits>::init(sb)` (27.78.2.2).

1 *Postcondition:* `rdbuf() == sb`.

`virtual ~basic_ostream();`

2 *Effects:* Destroys an object of class `basic_ostream`.

3 *Remarks:* Does not perform any operations on `rdbuf()`.

`basic_ostream(basic_ostream&& rhs);`

4 *Effects:* Move constructs from the rvalue `rhs`. This is accomplished by default constructing the base class and calling `basic_ios<charT, traits>::move(rhs)` to initialize the base class.

27.190.0.1 Class `basic_ostream` assign and swap [ostream.assign]

`basic_ostream& operator=(basic_ostream&& rhs);`

1 *Effects:* `swap(rhs)`.

2 *Returns:* `*this`.

`void swap(basic_ostream& rhs);`

3 *Effects:* Calls `basic_ios<charT, traits>::swap(rhs)`.

27.190.0.2 Class `basic_ostream::sentry` [ostream::sentry]

```
namespace std {
  template <class charT,class traits = char_traits<charT> >
  class basic_ostream<charT,traits>::sentry {
    bool ok_; // exposition only
  public:
    explicit sentry(basic_ostream<charT,traits>& os);
    ~sentry();
    explicit operator bool() const { return ok_; }

    sentry(const sentry&) = delete;
    sentry& operator=(const sentry&) = delete;
  };
}
```

1 The class `sentry` defines a class that is responsible for doing exception safe prefix and suffix operations.

`explicit sentry(basic_ostream<charT,traits>& os);`

2

If `os.good()` is nonzero, prepares for formatted or unformatted output. If `os.tie()` is not a null pointer, calls `os.tie()->flush()`.[324]

1

If, after any preparation is completed, `os.good()` is `true`, `ok_ == true` otherwise, `ok_ == false`. During preparation, the constructor may call `setstate(failbit)` (which may throw `ios_base::failure` (27.84.0.1))[325]

`~sentry();`

324) The call `os.tie()->flush()` does not necessarily occur if the function can determine that no synchronization is necessary.
325) The `sentry` constructor and destructor can also perform additional implementation-dependent operations.

1 If `((os.flags() & ios_base::unitbuf) && !uncaught_exception() && os.good())` is `true`, calls `os.rdbuf()->pubsync()`. If that function returns -1, sets `badbit` in `os.rdstate()` without propagating an exception.

```
explicit operator bool() const;
```

2 *Effects:* Returns `ok_`.

27.196.0.1 `basic_ostream` seek members [ostream.seeks]

1 Each seek member function begins execution by constructing an object of class `sentry`. It returns by destroying the `sentry` object.

```
pos_type tellp();
```

2 *Returns:* If `fail() != false`, returns `pos_type(-1)` to indicate failure. Otherwise, returns `rdbuf()->pubseekoff(0, cur, out)`.

```
basic_ostream<charT,traits>& seekp(pos_type pos);
```

3 *Effects:* If `fail() != true`, executes `rdbuf()->pubseekpos(pos, ios_base::out)`. In case of failure, the function calls `setstate(failbit)` (which may throw `ios_base::failure`).

1 *Returns:* `*this`.

```
basic_ostream<charT,traits>& seekp(off_type off, ios_base::seekdir dir);
```

2 *Effects:* If `fail() != true`, executes `rdbuf()->pubseekoff(off, dir, ios_base::out)`. In case of failure, the function calls `setstate(failbit)` (which may throw `ios_base::failure`).

3 *Returns:* `*this`.

27.198.0.1 Formatted output functions [ostream.formatted]

27.198.0.1.1 Common requirements [ostream.formatted.reqmts]

1

Each formatted output function begins execution by constructing an object of class `sentry`. If this object returns `true` when converted to a value of type `bool`, the function endeavors to generate the requested output. If the generation fails, then the formatted output function does `setstate(ios_base::failbit)`, which might throw an exception. If an exception is thrown during output, then `ios::badbit` is turned on[326] in `*this`'s error state. If `(exceptions()&badbit) != 0` then the exception is rethrown. Whether or not an exception is thrown, the `sentry` object is destroyed before leaving the formatted output function. If no exception is thrown, the result of the formatted output function is `*this`.

1 The descriptions of the individual formatted output functions describe how they perform output and do not mention the `sentry` object.

2

If a formatted output function of a stream `os` determines padding, it does so as follows. Given a `charT` character sequence `seq` where `charT` is the character type of the stream, if the length of `seq` is less than `os.width()`, then enough copies of `os.fill()` are added to this sequence as necessary to pad to a width of `os.width()` characters. If `(os.flags() & ios_base::adjustfield) == ios_base::left` is `true`, the fill characters are placed after the character sequence; otherwise, they are placed before the character sequence.

326) without causing an `ios::failure` to be thrown.

27.203.0.0.1 Arithmetic inserters [ostream.inserters.arithmetic]

```
operator<<(bool val);
operator<<(short val);
operator<<(unsigned short val);
operator<<(int val);
operator<<(unsigned int val);
operator<<(long val);
operator<<(unsigned long val);
operator<<(long long val);
operator<<(unsigned long long val);
operator<<(float val);
operator<<(double val);
operator<<(long double val);
operator<<(const void* val);
```

1

Effects: The classes `num_get<>` and `num_put<>` handle locale-dependent numeric formatting and parsing. These inserter functions use the imbued `locale` value to perform numeric formatting. When `val` is of type `bool`, `long`, `unsigned long`, `long long`, `unsigned long long`, `double`, `long double`, or `const void*`, the formatting conversion occurs as if it performed the following code fragment:

```
bool failed = use_facet<
  num_put<charT,ostreambuf_iterator<charT,traits> >
    >(getloc()).put(*this, *this, fill(), val).failed();
```

When `val` is of type `short` the formatting conversion occurs as if it performed the following code fragment:

```
ios_base::fmtflags baseflags = ios_base::flags() & ios_base::basefield;
bool failed = use_facet<
  num_put<charT,ostreambuf_iterator<charT,traits> >
    >(getloc()).put(*this, *this, fill(),
    baseflags == ios_base::oct || baseflags == ios_base::hex
      ? static_cast<long>(static_cast<unsigned short>(val))
      : static_cast<long>(val)).failed();
```

When `val` is of type `int` the formatting conversion occurs as if it performed the following code fragment:

```
ios_base::fmtflags baseflags = ios_base::flags() & ios_base::basefield;
bool failed = use_facet<
  num_put<charT,ostreambuf_iterator<charT,traits> >
    >(getloc()).put(*this, *this, fill(),
    baseflags == ios_base::oct || baseflags == ios_base::hex
      ? static_cast<long>(static_cast<unsigned int>(val))
      : static_cast<long>(val)).failed();
```

When `val` is of type `unsigned short` or `unsigned int` the formatting conversion occurs as if it performed the following code fragment:

```
bool failed = use_facet<
  num_put<charT,ostreambuf_iterator<charT,traits> >
    >(getloc()).put(*this, *this, fill(),
      static_cast<unsigned long>(val)).failed();
```

When `val` is of type `float` the formatting conversion occurs as if it performed the following code fragment:

```
bool failed = use_facet<
  num_put<charT,ostreambuf_iterator<charT,traits> >
    >(getloc()).put(*this, *this, fill(),
      static_cast<double>(val)).failed();
```

1 The first argument provides an object of the `ostreambuf_iterator<>` class which is an iterator for class `basic_ostream<>`. It bypasses `ostream`s and uses `streambuf`s directly. Class `locale` relies on these types as its interface to iostreams, since for flexibility it has been abstracted away from direct dependence on `ostream`. The second parameter is a reference to the base subobject of type `ios_base`. It provides formatting specifications such as field width, and a locale from which to obtain other facets. If `failed` is `true` then does `setstate(badbit)`, which may throw an exception, and returns.

2 *Returns:* `*this`.

27.210.0.0.1 `basic_ostream::operator<<` [ostream.inserters]

```
basic_ostream<charT,traits>& operator<<
    (basic_ostream<charT,traits>& (*pf)(basic_ostream<charT,traits>&));
```

1

Effects: None. Does not behave as a formatted output function (as described in 27.198.0.1.1).

1 *Returns:* `pf(*this)`.[327]

```
basic_ostream<charT,traits>& operator<<
    (basic_ios<charT,traits>& (*pf)(basic_ios<charT,traits>&));
```

2 *Effects:* Calls `pf(*this)`. This inserter does not behave as a formatted output function (as described in 27.198.0.1.1).

3 *Returns:* `*this`.[328]

```
basic_ostream<charT,traits>& operator<<
    (ios_base& (*pf)(ios_base&));
```

4 *Effects:* Calls `pf(*this)`. This inserter does not behave as a formatted output function (as described in 27.198.0.1.1).

5 *Returns:* `*this`.

```
basic_ostream<charT,traits>& operator<<
    (basic_streambuf<charT,traits>* sb);
```

6

Effects: Behaves as an unformatted output function (as described in 27.225.0.1, paragraph 1). After the sentry object is constructed, if `sb` is null calls `setstate(badbit)` (which may throw `ios_base::failure`).

1

Gets characters from `sb` and inserts them in `*this`. Characters are read from `sb` and inserted until any of the following occurs:

— end-of-file occurs on the input sequence;

— inserting in the output sequence fails (in which case the character to be inserted is not extracted);

— an exception occurs while getting a character from `sb`.

327) See, for example, the function signature `endl(basic_ostream&)` (27.234.0.1).
328) See, for example, the function signature `dec(ios_base&)` (27.91.1.3).

1

If the function inserts no characters, it calls `setstate(failbit)` (which may throw `ios_base::failure` (27.84.0.1)). If an exception was thrown while extracting a character, the function sets `failbit` in error state, and if `failbit` is on in `exceptions()` the caught exception is rethrown.

1 *Returns:* `*this`.

27.217.0.0.1 Character inserter function templates [ostream.inserters.character]

```
template<class charT, class traits>
  basic_ostream<charT,traits>& operator<<(basic_ostream<charT,traits>& out,
                                          charT c);
template<class charT, class traits>
  basic_ostream<charT,traits>& operator<<(basic_ostream<charT,traits>& out,
                                          char c);
  // specialization
template<class traits>
  basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>& out,
                                         char c);
  // signed and unsigned
template<class traits>
  basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>& out,
                                         signed char c);
template<class traits>
  basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>& out,
                                         unsigned char c);
```

1

Effects: Behaves as a formatted output function (27.198.0.1.1) of `out`. Constructs a character sequence `seq`. If `c` has type `char` and the character type of the stream is not `char`, then `seq` consists of `out.widen(c)`; otherwise `seq` consists of `c`. Determines padding for `seq` as described in 27.198.0.1.1. Inserts `seq` into `out`. Calls `os.width(0)`.

1

Returns: `out`.

```
template<class charT, class traits>
  basic_ostream<charT,traits>& operator<<(basic_ostream<charT,traits>& out,
                                          const charT* s);
template<class charT, class traits>
  basic_ostream<charT,traits>& operator<<(basic_ostream<charT,traits>& out,
                                          const char* s);
template<class traits>
  basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>& out,
                                         const char* s);
template<class traits>
  basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>& out,
                                         const signed char* s);
template<class traits>
  basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>& out,
                                         const unsigned char* s);
```

1 *Requires:* `s` shall not be a null pointer.

2

Effects: Behaves like a formatted inserter (as described in 27.198.0.1.1) of `out`. Creates a character sequence `seq` of `n` characters starting at `s`, each widened using `out.widen()` (27.83.0.1), where `n` is the number that would be computed as if by:

— `traits::length(s)` for the overload where the first argument is of type `basic_ostream<charT, traits>&` and the second is of type `const charT*`, and also for the overload where the first argument is of type `basic_ostream<char, traits>&` and the second is of type `const char*`,

— `std::char_traits<char>::length(s)` for the overload where the first argument is of type `basic_ostream<charT, traits>&` and the second is of type `const char*`,

— `traits::length(reinterpret_cast<const char*>(s))` for the other two overloads.

Determines padding for `seq` as described in 27.198.0.1.1. Inserts `seq` into `out`. Calls `width(0)`.

1 *Returns:* `out`.

27.225.0.1 Unformatted output functions [ostream.unformatted]

1

Each unformatted output function begins execution by constructing an object of class `sentry`. If this object returns `true`, while converting to a value of type `bool`, the function endeavors to generate the requested output. If an exception is thrown during output, then `ios::badbit` is turned on[329] in `*this`'s error state. If `(exceptions() & badbit) != 0` then the exception is rethrown. In any case, the unformatted output function ends by destroying the sentry object, then, if no exception was thrown, returning the value specified for the unformatted output function.

```
basic_ostream<charT,traits>& put(char_type c);
```

1 *Effects:* Behaves as an unformatted output function (as described in 27.225.0.1, paragraph 1). After constructing a sentry object, inserts the character `c`, if possible.[330]

2 Otherwise, calls `setstate(badbit)` (which may throw `ios_base::failure` (27.84.0.1)).

3

Returns: `*this`.

```
basic_ostream& write(const char_type* s, streamsize n);
```

1

Effects: Behaves as an unformatted output function (as described in 27.225.0.1, paragraph 1). After constructing a sentry object, obtains characters to insert from successive locations of an array whose first element is designated by `s`.[331] Characters are inserted until either of the following occurs:

— `n` characters are inserted;

— inserting in the output sequence fails (in which case the function calls `setstate(badbit)`, which may throw `ios_base::failure` (27.84.0.1)).

1 *Returns:* `*this`.

```
basic_ostream& flush();
```

329) without causing an `ios::failure` to be thrown.
330) Note that this function is not overloaded on types `signed char` and `unsigned char`.
331) Note that this function is not overloaded on types `signed char` and `unsigned char`.

2

Effects: Behaves as an unformatted output function (as described in 27.198.0.1.1, paragraph 1). If `rdbuf()` is not a null pointer, constructs a sentry object. If this object returns `true` when converted to a value of type `bool` the function calls `rdbuf()->pubsync()`. If that function returns -1 calls `setstate(badbit)` (which may throw `ios_base::failure` (27.84.0.1)). Otherwise, if the sentry object returns `false`, does nothing.

1 *Returns:* `*this`.

27.234.0.1 Standard `basic_ostream` manipulators [ostream.manip]

```
namespace std {
  template <class charT, class traits>
    basic_ostream<charT,traits>& endl(basic_ostream<charT,traits>& os);
}
```

1 *Effects:* Calls `os.put(os.widen('\n'))`, then `os.flush()`.

2 *Returns:* `os`.

```
namespace std {
  template <class charT, class traits>
    basic_ostream<charT,traits>& ends(basic_ostream<charT,traits>& os);
}
```

3

Effects: Inserts a null character into the output sequence: calls `os.put(charT())`.

1 *Returns:* `os`.

```
namespace std {
  template <class charT, class traits>
    basic_ostream<charT,traits>& flush(basic_ostream<charT,traits>& os);
}
```

2 *Effects:* Calls `os.flush()`.

3 *Returns:* `os`.

27.236.0.1 Rvalue stream insertion [ostream.rvalue]

```
template <class charT, class traits, class T>
  basic_ostream<charT, traits>&
  operator<<(basic_ostream<charT, traits>&& os, const T& x);
```

1 *Effects:* `os << x`

2 *Returns:* `os`

27.236.1 Standard manipulators [std.manip]

1 The header `<iomanip>` defines several functions that support extractors and inserters that alter information maintained by class `ios_base` and its derived classes.

```
unspecified  resetiosflags(ios_base::fmtflags mask);
```

2 *Returns:* An object of unspecified type such that if `out` is an object of type `basic_ostream<charT, traits>` then the expression `out << resetiosflags(mask)` behaves as if it called `f(out, mask)`, or if

in is an object of type basic_istream<charT, traits> then the expression in >> resetiosflags(mask) behaves as if it called f(in, mask), where the function f is defined as:[332]

```
void f(ios_base& str, ios_base::fmtflags mask) {
  // reset specified flags
  str.setf(ios_base::fmtflags(0), mask);
}
```

The expression out << resetiosflags(mask) shall have type basic_ostream<charT,traits>& and value out. The expression in >> resetiosflags(mask) shall have type basic_istream<charT, traits>& and value in.

unspecified setiosflags(ios_base::fmtflags mask);

3 *Returns:* An object of unspecified type such that if out is an object of type basic_ostream<charT, traits> then the expression out << setiosflags(mask) behaves as if it called f(out, mask), or if in is an object of type basic_istream<charT, traits> then the expression in >> setiosflags(mask) behaves as if it called f(in, mask), where the function f is defined as:

```
void f(ios_base& str, ios_base::fmtflags mask) {
  // set specified flags
  str.setf(mask);
}
```

The expression out << setiosflags(mask) shall have type basic_ostream<charT, traits>& and value out. The expression in >> setiosflags(mask) shall have type basic_istream<charT, traits>& and value in.

unspecified setbase(int base);

4 *Returns:* An object of unspecified type such that if out is an object of type basic_ostream<charT, traits> then the expression out << setbase(base) behaves as if it called f(out, base), or if in is an object of type basic_istream<charT, traits> then the expression in >> setbase(base) behaves as if it called f(in, base), where the function f is defined as:

```
void f(ios_base& str, int base) {
  // set basefield
  str.setf(base ==  8 ? ios_base::oct :
      base == 10 ? ios_base::dec :
      base == 16 ? ios_base::hex :
      ios_base::fmtflags(0), ios_base::basefield);
}
```

The expression out << setbase(base) shall have type basic_ostream<charT, traits>& and value out. The expression in >> setbase(base) shall have type basic_istream<charT, traits>& and value in.

unspecified setfill(char_type c);

5

Returns: An object of unspecified type such that if out is an object of type basic_ostream<charT, traits> and c has type charT then the expression out << setfill(c) behaves as if it called f(out, c), where the function f is defined as:

332) The expression cin >>resetiosflags(ios_base::skipws) clears ios_base::skipws in the format flags stored in the basic_istream<charT,traits> object cin (the same as cin >>noskipws), and the expression cout <<resetiosflags(ios_base::showbase) clears ios_base::showbase in the format flags stored in the basic_ostream<charT,traits> object cout (the same as cout <<noshowbase).

```
template<class charT, class traits>
void f(basic_ios<charT,traits>& str, charT c) {
  // set fill character
  str.fill(c);
}
```

The expression `out << setfill(c)` shall have type `basic_ostream<charT, traits>&` and value `out`.

unspecified `setprecision(int n);`

1 *Returns:* An object of unspecified type such that if `out` is an object of type `basic_ostream<charT, traits>` then the expression `out << setprecision(n)` behaves as if it called `f(out, n)`, or if `in` is an object of type `basic_istream<charT, traits>` then the expression `in >> setprecision(n)` behaves as if it called `f(in, n)`, where the function `f` is defined as:

```
void f(ios_base& str, int n) {
  // set precision
  str.precision(n);
}
```

The expression `out << setprecision(n)` shall have type `basic_ostream<charT, traits>&` and value `out`. The expression `in >> setprecision(n)` shall have type `basic_istream<charT, traits>&` and value `in`.

unspecified `setw(int n);`

2

Returns: An object of unspecified type such that if `out` is an instance of `basic_ostream<charT, traits>` then the expression `out << setw(n)` behaves as if it called `f(out, n)`, or if `in` is an object of type `basic_istream<charT, traits>` then the expression `in >> setw(n)` behaves as if it called `f(in, n)`, where the function `f` is defined as:

```
void f(ios_base& str, int n) {
  // set width
  str.width(n);
}
```

The expression `out << setw(n)` shall have type `basic_ostream<charT, traits>&` and value `out`. The expression `in >> setw(n)` shall have type `basic_istream<charT, traits>&` and value `in`.

27.242.1 Extended manipulators [ext.manip]

1 The header `<iomanip>` defines several functions that support extractors and inserters that allow for the parsing and formatting of sequences and values for money and time.

`template <class moneyT>` *unspecified* `get_money(moneyT& mon, bool intl = false);`

2 *Requires:* The type `moneyT` shall be either `long double` or a specialization of the `basic_string` template (Clause 21).

3 *Effects:* The expression `in >>get_money(mon, intl)` described below behaves as a formatted input function (27.128.0.1.1).

4 *Returns:* An object of unspecified type such that if `in` is an object of type `basic_istream<charT, traits>` then the expression `in >> get_money(mon, intl)` behaves as if it called `f(in, mon, intl)`, where the function `f` is defined as:

```
template <class charT, class traits, class moneyT>
void f(basic_ios<charT, traits>& str, moneyT& mon, bool intl) {
  typedef istreambuf_iterator<charT, traits> Iter;
  typedef money_get<charT, Iter> MoneyGet;

  ios_base::iostate err = ios_base::goodbit;
  const MoneyGet &mg = use_facet<MoneyGet>(str.getloc());

  mg.get(Iter(str.rdbuf()), Iter(), intl, str, err, mon);

  if (ios_base::goodbit != err)
    str.setstate(err);
}
```

The expression `in >> get_money(mon, intl)` shall have type `basic_istream<charT, traits>&` and value `in`.

`template <class moneyT> unspecified put_money(const moneyT& mon, bool intl = false);`

5 *Requires:* The type `moneyT` shall be either `long double` or a specialization of the `basic_string` template (Clause 21).

6 *Returns:* An object of unspecified type such that if `out` is an object of type `basic_ostream<charT, traits>` then the expression `out << put_money(mon, intl)` behaves as a formatted input function that calls `f(out, mon, intl)`, where the function `f` is defined as:

```
template <class charT, class traits, class moneyT>
void f(basic_ios<charT, traits>& str, const moneyT& mon, bool intl) {
  typedef ostreambuf_iterator<charT, traits> Iter;
  typedef money_put<charT, Iter> MoneyPut;

  const MoneyPut& mp = use_facet<MoneyPut>(str.getloc());
  const Iter end = mp.put(Iter(str.rdbuf()), intl, str, str.fill(), mon);

  if (end.failed())
    str.setstate(ios::badbit);
}
```

The expression `out << put_money(mon, intl)` shall have type `basic_ostream<charT, traits>&` and value `out`.

`template <class charT> unspecified get_time(struct tm* tmb, const charT* fmt);`

7 *Requires:* The argument `tmb` shall be a valid pointer to an object of type `struct tm`, and the argument `fmt` shall be a valid pointer to an array of objects of type `charT` with `char_traits<charT>::length(fmt)` elements.

8 *Returns:* An object of unspecified type such that if `in` is an object of type `basic_istream<charT, traits>` then the expression `in >> get_time(tmb, fmt)` behaves as if it called `f(in, tmb, fmt)`, where the function `f` is defined as:

```
template <class charT, class traits>
void f(basic_ios<charT, traits>& str, struct tm* tmb, const charT* fmt) {
  typedef istreambuf_iterator<charT, traits> Iter;
  typedef time_get<charT, Iter> TimeGet;

  ios_base::iostate err = ios_base::goodbit;
  const TimeGet& tg = use_facet<TimeGet>(str.getloc());
```

```
    tg.get(Iter(str.rdbuf()), Iter(), str, err, tmb,
      fmt, fmt + traits::length(fmt));

    if (err != ios_base::goodbit)
      str.setstate(err):
  }
```

The expression `in >> get_time(tmb, fmt)` shall have type `basic_istream<charT, traits>&` and value `in`.

```
template <class charT> unspecified  put_time(const struct tm* tmb, const charT* fmt);
```

9 *Requires:* The argument `tmb` shall be a valid pointer to an object of type `struct tm`, and the argument `fmt` shall be a valid pointer to an array of objects of type `charT` with `char_traits<charT>::length(fmt)` elements.

10 *Returns:* An object of unspecified type such that if `out` is an object of type `basic_ostream<charT, traits>` then the expression `out << put_time(tmb, fmt)` behaves as if it called `f(out, tmb, fmt)`, where the function `f` is defined as:

```
template <class charT, class traits>
void f(basic_ios<charT, traits>& str, const struct tm* tmb, const charT* fmt) {
  typedef ostreambuf_iterator<charT, traits> Iter;
  typedef time_put<charT, Iter> TimePut;

  const TimePut& tp = use_facet<TimePut>(str.getloc());
  const Iter end = tp.put(Iter(str.rdbuf()), str, str.fill(), tmb,
    fmt, fmt + traits::length(fmt));

  if (end.failed())
    str.setstate(ios_base::badbit);
}
```

The expression `out << put_time(tmb, fmt)` shall have type `basic_ostream<charT, traits>&` and value `out`.

27.242.2 Quoted manipulators [quoted.manip]

1 [*Note:* Quoted manipulators provide string insertion and extraction of quoted strings (for example, XML and CSV formats). Quoted manipulators are useful in ensuring that the content of a string with embedded spaces remains unchanged if inserted and then extracted via stream I/O. — *end note*]

```
template <class charT>
  unspecified  quoted(const charT* s, charT delim=charT('"'), charT escape=charT('\\'));
template <class charT, class traits, class Allocator>
  unspecified  quoted(const basic_string<charT, traits, Allocator>& s,
                      charT delim=charT('"'), charT escape=charT('\\'));
```

2 *Returns:* An object of unspecified type such that if `out` is an instance of `basic_ostream` with member type `char_type` the same as `charT` and with member type `traits_type`, which in the second form is the same as `traits`, then the expression `out << quoted(s, delim, escape)` behaves as a formatted output function (27.198.0.1.1) of `out`. This forms a character sequence `seq`, initially consisting of the following elements:

(2.1) — `delim`.

(2.2) — Each character in `s`. If the character to be output is equal to `escape` or `delim`, as determined by `traits_type::eq`, first output `escape`.

(2.3) — `delim`.

Let `x` be the number of elements initially in `seq`. Then padding is determined for `seq` as described in 27.198.0.1.1, `seq` is inserted as if by calling `out.rdbuf()->sputn(seq, n)`, where `n` is the larger of `out.width()` and `x`, and `out.width(0)` is called. The expression `out << quoted(s, delim, escape)` shall have type `basic_ostream<charT, traits>&` and value `out`.

```
template <class charT, class traits, class Allocator>
  unspecified  quoted(basic_string<charT, traits, Allocator>& s,
                      charT delim=charT('"'), charT escape=charT('\\'));
```

3 *Returns:* An object of unspecified type such that:

(3.1) — If `in` is an instance of `basic_istream` with member types `char_type` and `traits_type` the same as `charT` and `traits`, respectively, then the expression `in >> quoted(s, delim, escape)` behaves as if it extracts the following characters from `in` using `basic_istream::operator>>` (27.141.0.0.1) which may throw `ios_base::failure` (27.59.0.1.1):

(3.1.1) — If the first character extracted is equal to `delim`, as determined by `traits_type::eq`, then:

(3.1.1.1) — Turn off the `skipws` flag.

(3.1.1.2) — `s.clear()`

(3.1.1.3) — Until an unescaped `delim` character is reached or `!in`, extract characters from `in` and append them to `s`, except that if an `escape` is reached, ignore it and append the next character to `s`.

(3.1.1.4) — Discard the final `delim` character.

(3.1.1.5) — Restore the `skipws` flag to its original value.

(3.1.2) — Otherwise, `in >> s`.

(3.2) — If `out` is an instance of `basic_ostream` with member types `char_type` and `traits_type` the same as `charT` and `traits`, respectively, then the expression `out << quoted(s, delim, escape)` behaves as specified for the `const basic_string<charT, traits, Allocator>&` overload of the `quoted` function.

The expression `in >> quoted(s, delim, escape)` shall have type `basic_istream<charT, traits>&` and value `in`. The expression `out << quoted(s, delim, escape)` shall have type `basic_ostream <charT, traits>&` and value `out`.

27.243 String-based streams [string.streams]

27.243.1 Overview [string.streams.overview]

1 The header `<sstream>` defines four class templates and eight types that associate stream buffers with objects of class `basic_string`, as described in 21.26.

Header `<sstream>` synopsis

```
namespace std {
  template <class charT, class traits = char_traits<charT>,
        class Allocator = allocator<charT> >
    class basic_stringbuf;

  typedef basic_stringbuf<char>     stringbuf;
  typedef basic_stringbuf<wchar_t> wstringbuf;
```

```
  template <class charT, class traits = char_traits<charT>,
        class Allocator = allocator<charT> >
    class basic_istringstream;

  typedef basic_istringstream<char>     istringstream;
  typedef basic_istringstream<wchar_t> wistringstream;

  template <class charT, class traits = char_traits<charT>,
        class Allocator = allocator<charT> >
    class basic_ostringstream;
  typedef basic_ostringstream<char>     ostringstream;
  typedef basic_ostringstream<wchar_t> wostringstream;

  template <class charT, class traits = char_traits<charT>,
        class Allocator = allocator<charT> >
    class basic_stringstream;
  typedef basic_stringstream<char>     stringstream;
  typedef basic_stringstream<wchar_t> wstringstream;
}
```

27.243.2 Class template basic_stringbuf [stringbuf]

```
namespace std {
  template <class charT, class traits = char_traits<charT>,
      class Allocator = allocator<charT> >
  class basic_stringbuf : public basic_streambuf<charT,traits> {
  public:
    typedef charT                     char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;
    typedef traits                    traits_type;
    typedef Allocator                 allocator_type;

    // 27.243.2.1 Constructors:
    explicit basic_stringbuf(ios_base::openmode which
                             = ios_base::in | ios_base::out);
    explicit basic_stringbuf
    (const basic_string<charT,traits,Allocator>& str,
     ios_base::openmode which = ios_base::in | ios_base::out);
    basic_stringbuf(const basic_stringbuf& rhs) = delete;
    basic_stringbuf(basic_stringbuf&& rhs);

    // 27.244.0.1 Assign and swap:
    basic_stringbuf& operator=(const basic_stringbuf& rhs) = delete;
    basic_stringbuf& operator=(basic_stringbuf&& rhs);
    void swap(basic_stringbuf& rhs);

    // 27.245.0.1 Get and set:
    basic_string<charT,traits,Allocator> str() const;
    void str(const basic_string<charT,traits,Allocator>& s);

  protected:
    // 27.246.0.1 Overridden virtual functions:
    virtual int_type   underflow();
```

```
    virtual int_type   pbackfail(int_type c = traits::eof());
    virtual int_type   overflow (int_type c = traits::eof());
    virtual  basic_streambuf<charT,traits>* setbuf(charT*, streamsize);


    virtual pos_type seekoff(off_type off, ios_base::seekdir way,
                             ios_base::openmode which
                               = ios_base::in | ios_base::out);
    virtual pos_type seekpos(pos_type sp,
                             ios_base::openmode which
                               = ios_base::in | ios_base::out);

  private:
    ios_base::openmode mode;  // exposition only
  };

  template <class charT, class traits, class Allocator>
  void swap(basic_stringbuf<charT, traits, Allocator>& x,
            basic_stringbuf<charT, traits, Allocator>& y);
}
```

1 The class `basic_stringbuf` is derived from `basic_streambuf` to associate possibly the input sequence and possibly the output sequence with a sequence of arbitrary *characters*. The sequence can be initialized from, or made available as, an object of class `basic_string`.

2 For the sake of exposition, the maintained data is presented here as:

(2.1) — `ios_base::openmode mode`, has `in` set if the input sequence can be read, and `out` set if the output sequence can be written.

27.243.2.1 `basic_stringbuf` constructors [stringbuf.cons]

```
explicit basic_stringbuf(ios_base::openmode which =
                         ios_base::in | ios_base::out);
```

1 *Effects:* Constructs an object of class `basic_stringbuf`, initializing the base class with `basic_streambuf()` (27.96.0.1), and initializing `mode` with `which`.

2 *Postcondition:* `str() == ""`.

```
explicit basic_stringbuf(const basic_string<charT,traits,Allocator>& s,
                         ios_base::openmode which = ios_base::in | ios_base::out);
```

3 *Effects:* Constructs an object of class `basic_stringbuf`, initializing the base class with `basic_streambuf()` (27.96.0.1), and initializing `mode` with `which`. Then calls `str(s)`.

```
basic_stringbuf(basic_stringbuf&& rhs);
```

4

Effects: Move constructs from the rvalue `rhs`. It is implementation-defined whether the sequence pointers in `*this` (`eback()`, `gptr()`, `egptr()`, `pbase()`, `pptr()`, `epptr()`) obtain the values which `rhs` had. Whether they do or not, `*this` and `rhs` reference separate buffers (if any at all) after the construction. The openmode, locale and any other state of `rhs` is also copied.

1 *Postconditions:* Let `rhs_p` refer to the state of `rhs` just prior to this construction and let `rhs_a` refer to the state of `rhs` just after this construction.

(1.1) — `str() == rhs_p.str()`

(1.2) — `gptr() - eback() == rhs_p.gptr() - rhs_p.eback()`

(1.3) — `egptr() - eback() == rhs_p.egptr() - rhs_p.eback()`
(1.4) — `pptr() - pbase() == rhs_p.pptr() - rhs_p.pbase()`
(1.5) — `epptr() - pbase() == rhs_p.epptr() - rhs_p.pbase()`
(1.6) — `if (eback()) eback() != rhs_a.eback()`
(1.7) — `if (gptr()) gptr() != rhs_a.gptr()`
(1.8) — `if (egptr()) egptr() != rhs_a.egptr()`
(1.9) — `if (pbase()) pbase() != rhs_a.pbase()`
(1.10) — `if (pptr()) pptr() != rhs_a.pptr()`
(1.11) — `if (epptr()) epptr() != rhs_a.epptr()`

27.244.0.1 Assign and swap [stringbuf.assign]

```
basic_stringbuf& operator=(basic_stringbuf&& rhs);
```

1 *Effects:* After the move assignment `*this` has the observable state it would have had if it had been move constructed from `rhs` (see 27.243.2.1).

2 *Returns:* `*this`.

```
void swap(basic_stringbuf& rhs);
```

3 *Effects:* Exchanges the state of `*this` and `rhs`.

```
template <class charT, class traits, class Allocator>
void swap(basic_stringbuf<charT, traits, Allocator>& x,
          basic_stringbuf<charT, traits, Allocator>& y);
```

1 *Effects:* `x.swap(y)`.

27.245.0.1 Member functions [stringbuf.members]

```
basic_string<charT,traits,Allocator> str() const;
```

1 *Returns:* A `basic_string` object whose content is equal to the `basic_stringbuf` underlying character sequence. If the `basic_stringbuf` was created only in input mode, the resultant `basic_string` contains the character sequence in the range `[eback(),egptr())`. If the `basic_stringbuf` was created with `which & ios_base::out` being true then the resultant `basic_string` contains the character sequence in the range `[pbase(),high_mark)`, where `high_mark` represents the position one past the highest initialized character in the buffer. Characters can be initialized by writing to the stream, by constructing the `basic_stringbuf` with a `basic_string`, or by calling the `str(basic_string)` member function. In the case of calling the `str(basic_string)` member function, all characters initialized prior to the call are now considered uninitialized (except for those characters re-initialized by the new `basic_string`). Otherwise the `basic_stringbuf` has been created in neither input nor output mode and a zero length `basic_string` is returned.

```
void str(const basic_string<charT,traits,Allocator>& s);
```

1 *Effects:* Copies the content of `s` into the `basic_stringbuf` underlying character sequence and initializes the input and output sequences according to `mode`.

2 *Postconditions:* If `mode & ios_base::out` is true, `pbase()` points to the first underlying character and `epptr() >= pbase() + s.size()` holds; in addition, if `mode & ios_base::ate` is true, `pptr() == pbase() + s.size()` holds, otherwise `pptr() == pbase()` is true. If `mode & ios_base::in` is true, `eback()` points to the first underlying character, and both `gptr() == eback()` and `egptr() == eback() + s.size()` hold.

27.246.0.1 Overridden virtual functions [stringbuf.virtuals]

`int_type underflow();`

1 *Returns:* If the input sequence has a read position available, returns `traits::to_int_type(*gptr())`. Otherwise, returns `traits::eof()`. Any character in the underlying buffer which has been initialized is considered to be part of the input sequence.

`int_type pbackfail(int_type c = traits::eof());`

1 *Effects:* Puts back the character designated by `c` to the input sequence, if possible, in one of three ways:

(1.1) — If `traits::eq_int_type(c,traits::eof())` returns `false` and if the input sequence has a putback position available, and if `traits::eq(to_char_type(c),gptr()[-1])` returns `true`, assigns `gptr() - 1` to `gptr()`.
Returns: `c`.

(1.2) — If `traits::eq_int_type(c,traits::eof())` returns `false` and if the input sequence has a putback position available, and if `mode & ios_base::out` is nonzero, assigns `c` to `*--gptr()`.
Returns: `c`.

(1.3) — If `traits::eq_int_type(c,traits::eof())` returns `true` and if the input sequence has a putback position available, assigns `gptr() - 1` to `gptr()`.
Returns: `traits::not_eof(c)`.

2 *Returns:* `traits::eof()` to indicate failure.

3 *Remarks:* If the function can succeed in more than one of these ways, it is unspecified which way is chosen.

`int_type overflow(int_type c = traits::eof());`

4 *Effects:* Appends the character designated by `c` to the output sequence, if possible, in one of two ways:

(4.1) — If `traits::eq_int_type(c,traits::eof())` returns `false` and if either the output sequence has a write position available or the function makes a write position available (as described below), the function calls `sputc(c )`.
Signals success by returning `c`.

(4.2) — If `traits::eq_int_type(c,traits::eof())` returns `true`, there is no character to append.
Signals success by returning a value other than `traits::eof()`.

5 *Remarks:* The function can alter the number of write positions available as a result of any call.

6 *Returns:* `traits::eof()` to indicate failure.

7 The function can make a write position available only if `(mode & ios_base::out) != 0`. To make a write position available, the function reallocates (or initially allocates) an array object with a sufficient number of elements to hold the current array object (if any), plus at least one additional write position. If `(mode & ios_base::in) != 0`, the function alters the read end pointer `egptr()` to point just past the new write position.

```
pos_type seekoff(off_type off, ios_base::seekdir way,
                 ios_base::openmode which
                   = ios_base::in | ios_base::out);
```

Table 130 — `seekoff` positioning

Conditions	Result
`(which & ios_base::in) == ios_base::in`	positions the input sequence
`(which & ios_base::out) == ios_base::out`	positions the output sequence
`(which & (ios_base::in | ios_base::out)) == (ios_base::in) | ios_base::out))` and `way ==` either `ios_base::beg` or `ios_base::end`	positions both the input and the output sequences
Otherwise	the positioning operation fails.

8 *Effects:* Alters the stream position within one of the controlled sequences, if possible, as indicated in Table 130.

9 For a sequence to be positioned, if its next pointer (either `gptr()` or `pptr()`) is a null pointer and the new offset `newoff` is nonzero, the positioning operation fails. Otherwise, the function determines `newoff` as indicated in Table 131.

Table 131 — `newoff` values

Condition	`newoff` Value
`way == ios_base::beg`	0
`way == ios_base::cur`	the next pointer minus the beginning pointer (`xnext - xbeg`).
`way == ios_base::end`	the high mark pointer minus the beginning pointer (`high_mark - xbeg`).

10 If `(newoff + off) < 0`, or if `newoff + off` refers to an uninitialized character (as defined in 27.245.0.1 paragraph 1), the positioning operation fails. Otherwise, the function assigns `xbeg + newoff + off` to the next pointer `xnext`.

11 *Returns:* `pos_type(newoff)`, constructed from the resultant offset `newoff` (of type `off_type`), that stores the resultant stream position, if possible. If the positioning operation fails, or if the constructed object cannot represent the resultant stream position, the return value is `pos_type(off_type(-1))`.

```
pos_type seekpos(pos_type sp, ios_base::openmode which
                 = ios_base::in | ios_base::out);
```

12 *Effects:* Equivalent to `seekoff(off_type(sp), ios_base::beg, which)`.

13 *Returns:* `sp` to indicate success, or `pos_type(off_type(-1))` to indicate failure.

```
basic_streambuf<charT,traits>* setbuf(charT* s, streamsize n);
```

14 *Effects:* implementation-defined, except that `setbuf(0,0)` has no effect.

15 *Returns:* `this`.

27.247.1 Class template basic_istringstream [istringstream]

```
namespace std {
  template <class charT, class traits = char_traits<charT>,
        class Allocator = allocator<charT> >
  class basic_istringstream : public basic_istream<charT,traits> {
  public:
    typedef charT                     char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;
    typedef traits                    traits_type;
    typedef Allocator                 allocator_type;

    // 27.247.1.1 Constructors:
    explicit basic_istringstream(ios_base::openmode which = ios_base::in);
    explicit basic_istringstream(
               const basic_string<charT,traits,Allocator>& str,
               ios_base::openmode which = ios_base::in);
    basic_istringstream(const basic_istringstream& rhs) = delete;
    basic_istringstream(basic_istringstream&& rhs);

    // 27.247.1.2 Assign and swap:
    basic_istringstream& operator=(const basic_istringstream& rhs) = delete;
    basic_istringstream& operator=(basic_istringstream&& rhs);
    void swap(basic_istringstream& rhs);

    // 27.247.1.3 Members:
    basic_stringbuf<charT,traits,Allocator>* rdbuf() const;

    basic_string<charT,traits,Allocator> str() const;
    void str(const basic_string<charT,traits,Allocator>& s);
  private:
    basic_stringbuf<charT,traits,Allocator> sb; // exposition only
  };

  template <class charT, class traits, class Allocator>
  void swap(basic_istringstream<charT, traits, Allocator>& x,
            basic_istringstream<charT, traits, Allocator>& y);
}
```

1 The class basic_istringstream<charT, traits, Allocator> supports reading objects of class basic_string<charT, traits, Allocator>. It uses a basic_stringbuf<charT, traits, Allocator> object to control the associated storage. For the sake of exposition, the maintained data is presented here as:

(1.1) — sb, the stringbuf object.

27.247.1.1 basic_istringstream constructors [istringstream.cons]

```
explicit basic_istringstream(ios_base::openmode which = ios_base::in);
```

1 *Effects:* Constructs an object of class basic_istringstream<charT, traits>, initializing the base class with basic_istream(&sb) and initializing sb with basic_stringbuf<charT, traits, Allocator>(which | ios_base::in)) (27.243.2.1).

```
explicit basic_istringstream(
               const basic_string<charT, traits, Allocator>& str,
```

```
                ios_base::openmode which = ios_base::in);
```

2 *Effects:* Constructs an object of class `basic_istringstream<charT, traits>`, initializing the base class with `basic_istream(&sb)` and initializing `sb` with `basic_stringbuf<charT, traits, Allocator>(str, which | ios_base::in))` (27.243.2.1).

```
basic_istringstream(basic_istringstream&& rhs);
```

3 *Effects:* Move constructs from the rvalue `rhs`. This is accomplished by move constructing the base class, and the contained `basic_stringbuf`. Next `basic_istream<charT,traits>::set_rdbuf(&sb)` is called to install the contained `basic_stringbuf`.

27.247.1.2 Assign and swap [istringstream.assign]

```
basic_istringstream& operator=(basic_istringstream&& rhs);
```

1 *Effects:* Move assigns the base and members of `*this` from the base and corresponding members of `rhs`.

2 *Returns:* `*this`.

```
void swap(basic_istringstream& rhs);
```

3 *Effects:* Exchanges the state of `*this` and `rhs` by calling `basic_istream<charT,traits>::swap(rhs)` and `sb.swap(rhs.sb)`.

```
template <class charT, class traits, class Allocator>
void swap(basic_istringstream<charT, traits, Allocator>& x,
          basic_istringstream<charT, traits, Allocator>& y);
```

4 *Effects:* `x.swap(y)`.

27.247.1.3 Member functions [istringstream.members]

```
basic_stringbuf<charT,traits,Allocator>* rdbuf() const;
```

1 *Returns:* `const_cast<basic_stringbuf<charT,traits,Allocator>*>(&sb)`.

```
basic_string<charT,traits,Allocator> str() const;
```

2 *Returns:* `rdbuf()->str()`.

```
void str(const basic_string<charT,traits,Allocator>& s);
```

3 *Effects:* Calls `rdbuf()->str(s)`.

27.247.2 Class template basic_ostringstream [ostringstream]

```
namespace std {
  template <class charT, class traits = char_traits<charT>,
        class Allocator = allocator<charT> >
  class basic_ostringstream : public basic_ostream<charT,traits> {
  public:

    // types:
    typedef charT                     char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;
    typedef traits                    traits_type;
    typedef Allocator                 allocator_type;
```

```
    // 27.247.2.1 Constructors/destructor:
    explicit basic_ostringstream(ios_base::openmode which = ios_base::out);
    explicit basic_ostringstream(
             const basic_string<charT,traits,Allocator>& str,
             ios_base::openmode which = ios_base::out);
    basic_ostringstream(const basic_ostringstream& rhs) = delete;
    basic_ostringstream(basic_ostringstream&& rhs);

    // 27.247.2.2 Assign/swap:
    basic_ostringstream& operator=(const basic_ostringstream& rhs) = delete;
    basic_ostringstream& operator=(basic_ostringstream&& rhs);
    void swap(basic_ostringstream& rhs);

    // 27.247.2.3 Members:
    basic_stringbuf<charT,traits,Allocator>* rdbuf() const;

    basic_string<charT,traits,Allocator> str() const;
    void    str(const basic_string<charT,traits,Allocator>& s);
   private:
    basic_stringbuf<charT,traits,Allocator> sb; // exposition only
  };

  template <class charT, class traits, class Allocator>
  void swap(basic_ostringstream<charT, traits, Allocator>& x,
            basic_ostringstream<charT, traits, Allocator>& y);
}
```

1 The class `basic_ostringstream<charT, traits, Allocator>` supports writing objects of class `basic_string<charT, traits, Allocator>`. It uses a `basic_stringbuf` object to control the associated storage. For the sake of exposition, the maintained data is presented here as:

(1.1) — `sb`, the `stringbuf` object.

27.247.2.1 `basic_ostringstream` constructors [ostringstream.cons]

```
explicit basic_ostringstream(ios_base::openmode which = ios_base::out);
```

1 *Effects:* Constructs an object of class `basic_ostringstream`, initializing the base class with `basic_ostream(&sb)` and initializing `sb` with `basic_stringbuf<charT, traits, Allocator>(which | ios_base::out))` (27.243.2.1).

```
explicit basic_ostringstream(
  const basic_string<charT,traits,Allocator>& str,
  ios_base::openmode which = ios_base::out);
```

2 *Effects:* Constructs an object of class `basic_ostringstream<charT, traits>`, initializing the base class with `basic_ostream(&sb)` and initializing `sb` with `basic_stringbuf<charT, traits, Allocator>(str, which | ios_base::out))` (27.243.2.1).

```
basic_ostringstream(basic_ostringstream&& rhs);
```

3 *Effects:* Move constructs from the rvalue `rhs`. This is accomplished by move constructing the base class, and the contained `basic_stringbuf`. Next `basic_ostream<charT,traits>::set_rdbuf(&sb)` is called to install the contained `basic_stringbuf`.

27.247.2.2 Assign and swap [ostringstream.assign]

```
basic_ostringstream& operator=(basic_ostringstream&& rhs);
```

1 *Effects:* Move assigns the base and members of `*this` from the base and corresponding members of `rhs`.

2 *Returns:* `*this`.

```
void swap(basic_ostringstream& rhs);
```

3 *Effects:* Exchanges the state of `*this` and `rhs` by calling `basic_ostream<charT,traits>::swap(rhs)` and `sb.swap(rhs.sb)`.

```
template <class charT, class traits, class Allocator>
void swap(basic_ostringstream<charT, traits, Allocator>& x,
          basic_ostringstream<charT, traits, Allocator>& y);
```

4 *Effects:* `x.swap(y)`.

27.247.2.3 Member functions [ostringstream.members]

```
basic_stringbuf<charT,traits,Allocator>* rdbuf() const;
```

1 *Returns:* `const_cast<basic_stringbuf<charT,traits,Allocator>*>(&sb)`.

```
basic_string<charT,traits,Allocator> str() const;
```

2 *Returns:* `rdbuf()->str()`.

```
void str(const basic_string<charT,traits,Allocator>& s);
```

3 *Effects:* Calls `rdbuf()->str(s)`.

27.247.3 Class template `basic_stringstream` [stringstream]

```
namespace std {
  template <class charT, class traits = char_traits<charT>,
        class Allocator = allocator<charT> >
  class basic_stringstream
    : public basic_iostream<charT,traits> {
  public:

    // types:
    typedef charT                     char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;
    typedef traits                    traits_type;
    typedef Allocator                 allocator_type;

    // constructors/destructor
    explicit basic_stringstream(
    ios_base::openmode which = ios_base::out|ios_base::in);
    explicit basic_stringstream(
    const basic_string<charT,traits,Allocator>& str,
    ios_base::openmode which = ios_base::out|ios_base::in);
    basic_stringstream(const basic_stringstream& rhs) = delete;
    basic_stringstream(basic_stringstream&& rhs);
```

```
    // 27.247.3.2 Assign/swap:
    basic_stringstream& operator=(const basic_stringstream& rhs) = delete;
    basic_stringstream& operator=(basic_stringstream&& rhs);
    void swap(basic_stringstream& rhs);

    // Members:
    basic_stringbuf<charT,traits,Allocator>* rdbuf() const;
    basic_string<charT,traits,Allocator> str() const;
    void str(const basic_string<charT,traits,Allocator>& str);

  private:
    basic_stringbuf<charT, traits> sb;  // exposition only
  };

  template <class charT, class traits, class Allocator>
  void swap(basic_stringstream<charT, traits, Allocator>& x,
            basic_stringstream<charT, traits, Allocator>& y);
}
```

1 The class template `basic_stringstream<charT, traits>` supports reading and writing from objects of class `basic_string<charT, traits, Allocator>`. It uses a `basic_stringbuf<charT, traits, Allocator>` object to control the associated sequence. For the sake of exposition, the maintained data is presented here as

(1.1) — `sb`, the `stringbuf` object.

27.247.3.1 basic_stringstream constructors [stringstream.cons]

```
explicit basic_stringstream(
    ios_base::openmode which = ios_base::out|ios_base::in);
```

1 *Effects:* Constructs an object of class `basic_stringstream<charT,traits>`, initializing the base class with `basic_iostream(&sb)` and initializing `sb` with `basic_stringbuf<charT,traits,Allocator>(which)`.

```
explicit basic_stringstream(
    const basic_string<charT,traits,Allocator>& str,
    ios_base::openmode which = ios_base::out|ios_base::in);
```

2 *Effects:* Constructs an object of class `basic_stringstream<charT, traits>`, initializing the base class with `basic_iostream(&sb)` and initializing `sb` with `basic_stringbuf<charT, traits, Allocator>(str, which)`.

```
basic_stringstream(basic_stringstream&& rhs);
```

3 *Effects:* Move constructs from the rvalue `rhs`. This is accomplished by move constructing the base class, and the contained `basic_stringbuf`. Next `basic_istream<charT,traits>::set_rdbuf(&sb)` is called to install the contained `basic_stringbuf`.

27.247.3.2 Assign and swap [stringstream.assign]

```
basic_stringstream& operator=(basic_stringstream&& rhs);
```

1 *Effects:* Move assigns the base and members of `*this` from the base and corresponding members of `rhs`.

2 *Returns:* `*this`.

```
void swap(basic_stringstream& rhs);
```

3 *Effects:* Exchanges the state of `*this` and `rhs` by calling `basic_iostream<charT,traits>::swap(rhs)` and `sb.swap(rhs.sb)`.

```
template <class charT, class traits, class Allocator>
void swap(basic_stringstream<charT, traits, Allocator>& x,
          basic_stringstream<charT, traits, Allocator>& y);
```

4 *Effects:* `x.swap(y)`.

27.247.3.3 Member functions [stringstream.members]

```
basic_stringbuf<charT,traits,Allocator>* rdbuf() const;
```

1 *Returns:* `const_cast<basic_stringbuf<charT,traits,Allocator>*>(&sb)`

```
basic_string<charT,traits,Allocator> str() const;
```

2 *Returns:* `rdbuf()->str()`.

```
void str(const basic_string<charT,traits,Allocator>& str);
```

3 *Effects:* Calls `rdbuf()->str(str)`.

27.248 File-based streams [file.streams]

27.248.1 File streams [fstreams]

1 The header `<fstream>` defines four class templates and eight types that associate stream buffers with files and assist reading and writing files.

Header `<fstream>` synopsis

```
namespace std {
  template <class charT, class traits = char_traits<charT> >
    class basic_filebuf;
  typedef basic_filebuf<char>    filebuf;
  typedef basic_filebuf<wchar_t> wfilebuf;

  template <class charT, class traits = char_traits<charT> >
    class basic_ifstream;
  typedef basic_ifstream<char>    ifstream;
  typedef basic_ifstream<wchar_t> wifstream;

  template <class charT, class traits = char_traits<charT> >
    class basic_ofstream;
  typedef basic_ofstream<char>    ofstream;
  typedef basic_ofstream<wchar_t> wofstream;

  template <class charT, class traits = char_traits<charT> >
    class basic_fstream;
  typedef basic_fstream<char>     fstream;
  typedef basic_fstream<wchar_t> wfstream;
}
```

2 In this subclause, the type name `FILE` refers to the type `FILE` declared in `<cstdio>` (27.310.1).

3 [*Note:* The class template `basic_filebuf` treats a file as a source or sink of bytes. In an environment that uses a large character set, the file typically holds multibyte character sequences and the `basic_filebuf` object converts those multibyte sequences into wide character sequences. — *end note*]

27.248.1.1 Class template basic_filebuf [filebuf]

```
namespace std {
  template <class charT, class traits = char_traits<charT> >
  class basic_filebuf : public basic_streambuf<charT,traits> {
  public:
    typedef charT                     char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;
    typedef traits                    traits_type;

    // 27.263.0.1 Constructors/destructor:
    basic_filebuf();
    basic_filebuf(const basic_filebuf& rhs) = delete;
    basic_filebuf(basic_filebuf&& rhs);
    virtual ~basic_filebuf();

    // 27.269.0.1 Assign/swap:
    basic_filebuf& operator=(const basic_filebuf& rhs) = delete;
    basic_filebuf& operator=(basic_filebuf&& rhs);
    void swap(basic_filebuf& rhs);

     // 27.269.0.2 Members:
    bool is_open() const;
    basic_filebuf<charT,traits>* open(const char* s,
        ios_base::openmode mode);
    basic_filebuf<charT,traits>* open(const string& s,
        ios_base::openmode mode);
    basic_filebuf<charT,traits>* close();

  protected:
    // 27.288.0.1 Overridden virtual functions:
    virtual streamsize showmanyc();
    virtual int_type underflow();
    virtual int_type uflow();
    virtual int_type pbackfail(int_type c = traits::eof());
    virtual int_type overflow (int_type c = traits::eof());

    virtual basic_streambuf<charT,traits>*
        setbuf(char_type* s, streamsize n);
    virtual pos_type seekoff(off_type off, ios_base::seekdir way,
        ios_base::openmode which = ios_base::in | ios_base::out);
    virtual pos_type seekpos(pos_type sp,
        ios_base::openmode which = ios_base::in | ios_base::out);
    virtual int      sync();
    virtual void     imbue(const locale& loc);
  };

  template <class charT, class traits>
  void swap(basic_filebuf<charT, traits>& x,
            basic_filebuf<charT, traits>& y);
}
```

1

The class basic_filebuf<charT,traits> associates both the input sequence and the output sequence with a file.

1

The restrictions on reading and writing a sequence controlled by an object of class basic_filebuf<charT, traits> are the same as for reading and writing with the Standard C library FILEs.

1

In particular:

— If the file is not open for reading the input sequence cannot be read.

— If the file is not open for writing the output sequence cannot be written.

— A joint file position is maintained for both the input sequence and the output sequence.

1 An instance of basic_filebuf behaves as described in 27.248.1.1 provided traits::pos_type is fpos<traits::state_type>. Otherwise the behavior is undefined.

2 In order to support file I/O and multibyte/wide character conversion, conversions are performed using members of a facet, referred to as a_codecvt in following sections, obtained as if by

```
const codecvt<charT,char,typename traits::state_type>& a_codecvt =
  use_facet<codecvt<charT,char,typename traits::state_type> >(getloc());
```

27.263.0.1 basic_filebuf constructors [filebuf.cons]

basic_filebuf();

1 *Effects:* Constructs an object of class basic_filebuf<charT,traits>, initializing the base class with basic_streambuf<charT,traits>() (27.96.0.1).

2 *Postcondition:* is_open() == false.

basic_filebuf(basic_filebuf&& rhs);

3 *Effects:* Move constructs from the rvalue rhs. It is implementation-defined whether the sequence pointers in *this (eback(), gptr(), egptr(), pbase(), pptr(), epptr()) obtain the values which rhs had. Whether they do or not, *this and rhs reference separate buffers (if any at all) after the construction. Additionally *this references the file which rhs did before the construction, and rhs references no file after the construction. The openmode, locale and any other state of rhs is also copied.

4 *Postconditions:* Let rhs_p refer to the state of rhs just prior to this construction and let rhs_a refer to the state of rhs just after this construction.

(4.1) — is_open() == rhs_p.is_open()

(4.2) — rhs_a.is_open() == false

(4.3) — gptr() - eback() == rhs_p.gptr() - rhs_p.eback()

(4.4) — egptr() - eback() == rhs_p.egptr() - rhs_p.eback()

(4.5) — pptr() - pbase() == rhs_p.pptr() - rhs_p.pbase()

(4.6) — epptr() - pbase() == rhs_p.epptr() - rhs_p.pbase()

(4.7) — if (eback()) eback() != rhs_a.eback()

(4.8) — if (gptr()) gptr() != rhs_a.gptr()

(4.9) — if (egptr()) egptr() != rhs_a.egptr()

(4.10) — `if (pbase()) pbase() != rhs_a.pbase()`

(4.11) — `if (pptr()) pptr() != rhs_a.pptr()`

(4.12) — `if (epptr()) epptr() != rhs_a.epptr()`

```
virtual ~basic_filebuf();
```

5

Effects: Destroys an object of class `basic_filebuf<charT,traits>`. Calls `close()`. If an exception occurs during the destruction of the object, including the call to `close()`, the exception is caught but not rethrown (see 17.677.0.1).

27.269.0.1 Assign and swap [filebuf.assign]

```
basic_filebuf& operator=(basic_filebuf&& rhs);
```

1 *Effects:* Calls `this->close()` then move assigns from `rhs`. After the move assignment `*this` has the observable state it would have had if it had been move constructed from `rhs` (see 27.263.0.1).

2 *Returns:* `*this`.

```
void swap(basic_filebuf& rhs);
```

3 *Effects:* Exchanges the state of `*this` and `rhs`.

```
template <class charT, class traits>
void swap(basic_filebuf<charT, traits>& x,
          basic_filebuf<charT, traits>& y);
```

4 *Effects:* `x.swap(y)`.

27.269.0.2 Member functions [filebuf.members]

```
bool is_open() const;
```

1

Returns: `true` if a previous call to `open` succeeded (returned a non-null value) and there has been no intervening call to close.

```
basic_filebuf<charT,traits>* open(const char* s,
    ios_base::openmode mode);
```

1

Effects: If `is_open() != false`, returns a null pointer. Otherwise, initializes the `filebuf` as required. It then opens a file, if possible, whose name is the NTBS `s` (as if by calling `std::fopen(s,modstr)`). The NTBS `modstr` is determined from `mode & ~ios_base::ate` as indicated in Table 132. If `mode` is not some combination of flags shown in the table then the open fails.

1

If the open operation succeeds and `(mode & ios_base::ate) != 0`, positions the file to the end (as if by calling `std::fseek(file,0,SEEK_END)`).[333]

1 If the repositioning operation fails, calls `close()` and returns a null pointer to indicate failure.

2

Returns: `this` if successful, a null pointer otherwise.

333) The macro `SEEK_END` is defined, and the function signatures `fopen(const char*, const char*)` and `fseek(FILE*, long, int)` are declared, in `<cstdio>` (27.310.1).

Table 132 — File open modes

ios_base flag combination					stdio equivalent
binary	in	out	trunc	app	
		+			"w"
		+		+	"a"
				+	"a"
		+	+		"w"
	+				"r"
	+	+			"r+"
	+	+	+		"w+"
	+	+		+	"a+"
	+			+	"a+"
+		+			"wb"
+		+		+	"ab"
+				+	"ab"
+		+	+		"wb"
+	+				"rb"
+	+	+			"r+b"
+	+	+	+		"w+b"
+	+	+		+	"a+b"
+	+			+	"a+b"

```
basic_filebuf<charT,traits>* open(const string& s,
    ios_base::openmode mode);
```

Returns: `open(s.c_str(), mode);`

```
basic_filebuf<charT,traits>* close();
```

1 *Effects:* If `is_open() == false`, returns a null pointer. If a put area exists, calls `overflow(traits::eof())` to flush characters. If the last virtual member function called on `*this` (between `underflow`, `overflow`, `seekoff`, and `seekpos`) was `overflow` then calls `a_codecvt.unshift` (possibly several times) to determine a termination sequence, inserts those characters and calls `overflow(traits::eof())` again. Finally, regardless of whether any of the preceding calls fails or throws an exception, the function closes the file (as if by calling `std::fclose(file)`).[334] If any of the calls made by the function, including `std::fclose`, fails, `close` fails by returning a null pointer. If one of these calls throws an exception, the exception is caught and rethrown after closing the file.

1 *Returns:* `this` on success, a null pointer otherwise.

2 *Postcondition:* `is_open() == false`.

27.288.0.1 Overridden virtual functions [filebuf.virtuals]

```
streamsize showmanyc();
```

1 *Effects:* Behaves the same as `basic_streambuf::showmanyc()` (27.99.0.2).

2 *Remarks:* An implementation might well provide an overriding definition for this function signature if it can determine that more characters can be read from the input sequence.

334) The function signature `fclose(FILE*)` is declared in `<cstdio>` (27.310.1).

`int_type underflow();`

1 *Effects:* Behaves according to the description of `basic_streambuf<charT,traits>::underflow()`, with the specialization that a sequence of characters is read from the input sequence as if by reading from the associated file into an internal buffer (`extern_buf`) and then as if by doing

```
char   extern_buf[XSIZE];
char*  extern_end;
charT  intern_buf[ISIZE];
charT* intern_end;
codecvt_base::result r =
  a_codecvt.in(state, extern_buf, extern_buf+XSIZE, extern_end,
               intern_buf, intern_buf+ISIZE, intern_end);
```

This shall be done in such a way that the class can recover the position (`fpos_t`) corresponding to each character between `intern_buf` and `intern_end`. If the value of `r` indicates that `a_codecvt.in()` ran out of space in `intern_buf`, retry with a larger `intern_buf`.

`int_type uflow();`

2

Effects: Behaves according to the description of `basic_streambuf<charT,traits>::uflow()`, with the specialization that a sequence of characters is read from the input with the same method as used by `underflow`.

`int_type pbackfail(int_type c = traits::eof());`

1 *Effects:* Puts back the character designated by `c` to the input sequence, if possible, in one of three ways:

(1.1) — If `traits::eq_int_type(c,traits::eof())` returns `false` and if the function makes a putback position available and if `traits::eq(to_char_type(c),gptr()[-1])` returns `true`, decrements the next pointer for the input sequence, `gptr()`.

Returns: `c`.

(1.2) — If `traits::eq_int_type(c,traits::eof())` returns `false` and if the function makes a putback position available and if the function is permitted to assign to the putback position, decrements the next pointer for the input sequence, and stores `c` there.

Returns: `c`.

(1.3) — If `traits::eq_int_type(c,traits::eof())` returns `true`, and if either the input sequence has a putback position available or the function makes a putback position available, decrements the next pointer for the input sequence, `gptr()`.

Returns: `traits::not_eof(c)`.

2 *Returns:* `traits::eof()` to indicate failure.

3 *Remarks:* If `is_open() == false`, the function always fails.

4 The function does not put back a character directly to the input sequence.

5 If the function can succeed in more than one of these ways, it is unspecified which way is chosen. The function can alter the number of putback positions available as a result of any call.

`int_type overflow(int_type c = traits::eof());`

6 *Effects:* Behaves according to the description of `basic_streambuf<charT,traits>::overflow(c)`, except that the behavior of "consuming characters" is performed by first converting as if by:

```
charT* b = pbase();
charT* p = pptr();
charT* end;
char   xbuf[XSIZE];
char*  xbuf_end;
codecvt_base::result r =
  a_codecvt.out(state, b, p, end, xbuf, xbuf+XSIZE, xbuf_end);
```

and then

(6.1) — If `r == codecvt_base::error` then fail.

(6.2) — If `r == codecvt_base::noconv` then output characters from `b` up to (and not including) `p`.

(6.3) — If `r == codecvt_base::partial` then output to the file characters from `xbuf` up to `xbuf_end`, and repeat using characters from `end` to `p`. If output fails, fail (without repeating).

(6.4) — Otherwise output from `xbuf` to `xbuf_end`, and fail if output fails. At this point if `b != p` and `b == end` (`xbuf` isn't large enough) then increase `XSIZE` and repeat from the beginning.

7 *Returns:* `traits::not_eof(c)` to indicate success, and `traits::eof()` to indicate failure. If `is_open() == false`, the function always fails.

```
basic_streambuf* setbuf(char_type* s, streamsize n);
```

8

Effects: If `setbuf(0,0)` is called on a stream before any I/O has occurred on that stream, the stream becomes unbuffered. Otherwise the results are implementation-defined. "Unbuffered" means that `pbase()` and `pptr()` always return null and output to the file should appear as soon as possible.

```
pos_type seekoff(off_type off, ios_base::seekdir way,
    ios_base::openmode which = ios_base::in | ios_base::out);
```

1

Effects: Let `width` denote `a_codecvt.encoding()`. If `is_open() == false`, or `off != 0 && width <= 0`, then the positioning operation fails. Otherwise, if `way != basic_ios::cur` or `off != 0`, and if the last operation was output, then update the output sequence and write any unshift sequence. Next, seek to the new position: if `width > 0`, call `std::fseek(file, width * off, whence)`, otherwise call `std::fseek(file, 0, whence)`.

1 *Remarks:* "The last operation was output" means either the last virtual operation was overflow or the put buffer is non-empty. "Write any unshift sequence" means, if `width` if less than zero then call `a_codecvt.unshift(state, xbuf, xbuf+XSIZE, xbuf_end)` and output the resulting unshift sequence. The function determines one of three values for the argument `whence`, of type `int`, as indicated in Table 133.

Table 133 — `seekoff` effects

`way` Value	`stdio` Equivalent
`basic_ios::beg`	`SEEK_SET`
`basic_ios::cur`	`SEEK_CUR`
`basic_ios::end`	`SEEK_END`

2 *Returns:* A newly constructed `pos_type` object that stores the resultant stream position, if possible. If the positioning operation fails, or if the object cannot represent the resultant stream position, returns `pos_type(off_type(-1))`.

```
pos_type seekpos(pos_type sp,
    ios_base::openmode which = ios_base::in | ios_base::out);
```

3 Alters the file position, if possible, to correspond to the position stored in `sp` (as described below). Altering the file position performs as follows:

1. if `(om & ios_base::out) != 0`, then update the output sequence and write any unshift sequence;
2. set the file position to `sp`;
3. if `(om & ios_base::in) != 0`, then update the input sequence;

where `om` is the open mode passed to the last call to `open()`. The operation fails if `is_open()` returns false.

4 If `sp` is an invalid stream position, or if the function positions neither sequence, the positioning operation fails. If `sp` has not been obtained by a previous successful call to one of the positioning functions (`seekoff` or `seekpos`) on the same file the effects are undefined.

5 *Returns:* `sp` on success. Otherwise returns `pos_type(off_type(-1))`.

```
int sync();
```

6 *Effects:* If a put area exists, calls `filebuf::overflow` to write the characters to the file. If a get area exists, the effect is implementation-defined.

```
void imbue(const locale& loc);
```

7 *Requires:* If the file is not positioned at its beginning and the encoding of the current locale as determined by `a_codecvt.encoding()` is state-dependent (22.41.0.0.1) then that facet is the same as the corresponding facet of `loc`.

8 *Effects:* Causes characters inserted or extracted after this call to be converted according to `loc` until another call of `imbue`.

9 *Remark:* This may require reconversion of previously converted characters. This in turn may require the implementation to be able to reconstruct the original contents of the file.

27.292.0.1 Class template `basic_ifstream` [ifstream]

```
namespace std {
  template <class charT, class traits = char_traits<charT> >
  class basic_ifstream : public basic_istream<charT,traits> {
  public:
    typedef charT                     char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;
    typedef traits                    traits_type;

    // 27.292.0.2 Constructors:
    basic_ifstream();
    explicit basic_ifstream(const char* s,
        ios_base::openmode mode = ios_base::in);
    explicit basic_ifstream(const string& s,
        ios_base::openmode mode = ios_base::in);
    basic_ifstream(const basic_ifstream& rhs) = delete;
    basic_ifstream(basic_ifstream&& rhs);

    // 27.297.0.1 Assign/swap:
```

```
    basic_ifstream& operator=(const basic_ifstream& rhs) = delete;
    basic_ifstream& operator=(basic_ifstream&& rhs);
    void swap(basic_ifstream& rhs);

    // 27.297.0.2 Members:
    basic_filebuf<charT,traits>* rdbuf() const;

    bool is_open() const;
    void open(const char* s, ios_base::openmode mode = ios_base::in);
    void open(const string& s, ios_base::openmode mode = ios_base::in);
    void close();
  private:
    basic_filebuf<charT,traits> sb; // exposition only
  };

  template <class charT, class traits>
  void swap(basic_ifstream<charT, traits>& x,
            basic_ifstream<charT, traits>& y);
}
```

1 The class `basic_ifstream<charT, traits>` supports reading from named files. It uses a `basic_filebuf<charT, traits>` object to control the associated sequence. For the sake of exposition, the maintained data is presented here as:

(1.1) — `sb`, the `filebuf` object.

27.292.0.2 `basic_ifstream` constructors [ifstream.cons]

```
basic_ifstream();
```

1 *Effects:* Constructs an object of class `basic_ifstream<charT,traits>`, initializing the base class with `basic_istream(&sb)` and initializing `sb` with `basic_filebuf<charT,traits>()` (27.116.0.0.1, 27.263.0.1).

```
explicit basic_ifstream(const char* s,
    ios_base::openmode mode = ios_base::in);
```

1 *Effects:* Constructs an object of class `basic_ifstream`, initializing the base class with `basic_istream(&sb)` and initializing `sb` with `basic_filebuf<charT, traits>()` (27.116.0.0.1, 27.263.0.1), then calls `rdbuf()->open(s, mode | ios_base::in)`. If that function returns a null pointer, calls `setstate(failbit)`.

```
explicit basic_ifstream(const string& s,
    ios_base::openmode mode = ios_base::in);
```

1 *Effects:* the same as `basic_ifstream(s.c_str(), mode)`.

```
basic_ifstream(basic_ifstream&& rhs);
```

2 *Effects:* Move constructs from the rvalue `rhs`. This is accomplished by move constructing the base class, and the contained `basic_filebuf`. Next `basic_istream<charT,traits>::set_rdbuf(&sb)` is called to install the contained `basic_filebuf`.

27.297.0.1 Assign and swap [ifstream.assign]

```
basic_ifstream& operator=(basic_ifstream&& rhs);
```

1 *Effects:* Move assigns the base and members of `*this` from the base and corresponding members of `rhs`.

2 *Returns:* `*this`.

```
void swap(basic_ifstream& rhs);
```

3 *Effects:* Exchanges the state of `*this` and `rhs` by calling `basic_istream<charT,traits>::swap(rhs)` and `sb.swap(rhs.sb)`.

```
template <class charT, class traits>
void swap(basic_ifstream<charT, traits>& x,
          basic_ifstream<charT, traits>& y);
```

4 *Effects:* `x.swap(y)`.

27.297.0.2 Member functions [ifstream.members]

```
basic_filebuf<charT,traits>* rdbuf() const;
```

1 *Returns:* `const_cast<basic_filebuf<charT,traits>*>(&sb)`.

```
bool is_open() const;
```

2

Returns: `rdbuf()->is_open()`.

```
void open(const char* s, ios_base::openmode mode = ios_base::in);
```

1

Effects: Calls `rdbuf()->open(s, mode | ios_base::in)`. If that function does not return a null pointer calls `clear()`, otherwise calls `setstate(failbit)` (which may throw `ios_base::failure` (27.84.0.1)).

```
void open(const string& s, ios_base::openmode mode = ios_base::in);
```

1 *Effects:* calls `open(s.c_str(), mode)`.

```
void close();
```

2

Effects: Calls `rdbuf()->close()` and, if that function returns a null pointer, calls `setstate(failbit)` (which may throw `ios_base::failure` (27.84.0.1)).

27.303.0.1 Class template `basic_ofstream` [ofstream]

```
namespace std {
  template <class charT, class traits = char_traits<charT> >
  class basic_ofstream : public basic_ostream<charT,traits> {
  public:
    typedef charT                     char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;
    typedef traits                    traits_type;
```

```
    // 27.303.0.2 Constructors:
    basic_ofstream();
    explicit basic_ofstream(const char* s,
        ios_base::openmode mode = ios_base::out);
    explicit basic_ofstream(const string& s,
        ios_base::openmode mode = ios_base::out);
    basic_ofstream(const basic_ofstream& rhs) = delete;
    basic_ofstream(basic_ofstream&& rhs);

    // 27.306.0.1 Assign/swap:
    basic_ofstream& operator=(const basic_ofstream& rhs) = delete;
    basic_ofstream& operator=(basic_ofstream&& rhs);
    void swap(basic_ofstream& rhs);

    // 27.306.0.2 Members:
    basic_filebuf<charT,traits>* rdbuf() const;

    bool is_open() const;
    void open(const char* s, ios_base::openmode mode = ios_base::out);
    void open(const string& s, ios_base::openmode mode = ios_base::out);
    void close();
  private:
    basic_filebuf<charT,traits> sb; // exposition only
  };

  template <class charT, class traits>
  void swap(basic_ofstream<charT, traits>& x,
            basic_ofstream<charT, traits>& y);
}
```

1 The class `basic_ofstream<charT, traits>` supports writing to named files. It uses a `basic_filebuf<charT, traits>` object to control the associated sequence. For the sake of exposition, the maintained data is presented here as:

(1.1) — `sb`, the `filebuf` object.

27.303.0.2 `basic_ofstream` constructors [ofstream.cons]

`basic_ofstream();`

1 *Effects:* Constructs an object of class `basic_ofstream<charT,traits>`, initializing the base class with `basic_ostream(&sb)` and initializing `sb` with `basic_filebuf<charT,traits>()` (27.188.0.1, 27.263.0.1).

```
explicit basic_ofstream(const char* s,
    ios_base::openmode mode = ios_base::out);
```

1 *Effects:* Constructs an object of class `basic_ofstream<charT,traits>`, initializing the base class with `basic_ostream(&sb)` and initializing `sb` with `basic_filebuf<charT,traits>()` (27.188.0.1, 27.263.0.1), then calls `rdbuf()->open(s, mode|ios_base::out)`. If that function returns a null pointer, calls `setstate(failbit)`.

```
explicit basic_ofstream(const string& s,
    ios_base::openmode mode = ios_base::out);
```

1 *Effects:* the same as `basic_ofstream(s.c_str(), mode);`

```
basic_ofstream(basic_ofstream&& rhs);
```

2 *Effects:* Move constructs from the rvalue `rhs`. This is accomplished by move constructing the base class, and the contained `basic_filebuf`. Next `basic_ostream<charT,traits>::set_rdbuf(&sb)` is called to install the contained `basic_filebuf`.

27.306.0.1 Assign and swap [ofstream.assign]

```
basic_ofstream& operator=(basic_ofstream&& rhs);
```

1 *Effects:* Move assigns the base and members of `*this` from the base and corresponding members of `rhs`.

2 *Returns:* `*this`.

```
void swap(basic_ofstream& rhs);
```

3 *Effects:* Exchanges the state of `*this` and `rhs` by calling `basic_ostream<charT,traits>::swap(rhs)` and `sb.swap(rhs.sb)`.

```
template <class charT, class traits>
void swap(basic_ofstream<charT, traits>& x,
          basic_ofstream<charT, traits>& y);
```

4 *Effects:* `x.swap(y)`.

27.306.0.2 Member functions [ofstream.members]

```
basic_filebuf<charT,traits>* rdbuf() const;
```

1 *Returns:* `const_cast<basic_filebuf<charT,traits>*>(&sb)`.

```
bool is_open() const;
```

2

Returns: `rdbuf()->is_open()`.

```
void open(const char* s, ios_base::openmode mode = ios_base::out);
```

1

Effects: Calls `rdbuf()->open(s, mode | ios_base::out)`. If that function does not return a null pointer calls `clear()`, otherwise calls `setstate(failbit)` (which may throw `ios_base::failure` (27.84.0.1)).

```
void close();
```

1 *Effects:* Calls `rdbuf()->close()` and, if that function fails (returns a null pointer), calls `setstate(failbit)` (which may throw `ios_base::failure` (27.84.0.1)).

```
void open(const string& s, ios_base::openmode mode = ios_base::out);
```

2 *Effects:* calls `open(s.c_str(), mode);`

27.308.0.1 Class template basic_fstream [fstream]

```
namespace std {
  template <class charT, class traits=char_traits<charT> >
  class basic_fstream
    : public basic_iostream<charT,traits> {

  public:
    typedef charT                     char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;
    typedef traits                    traits_type;

    // constructors/destructor
    basic_fstream();
    explicit basic_fstream(const char* s,
        ios_base::openmode mode = ios_base::in|ios_base::out);
    explicit basic_fstream(const string& s,
        ios_base::openmode mode = ios_base::in|ios_base::out);
    basic_fstream(const basic_fstream& rhs) = delete;
    basic_fstream(basic_fstream&& rhs);

    // 27.308.0.3 Assign/swap:
    basic_fstream& operator=(const basic_fstream& rhs) = delete;
    basic_fstream& operator=(basic_fstream&& rhs);
    void swap(basic_fstream& rhs);

    // Members:
    basic_filebuf<charT,traits>* rdbuf() const;
    bool is_open() const;
    void open(const char* s,
        ios_base::openmode mode = ios_base::in|ios_base::out);
    void open(const string& s,
        ios_base::openmode mode = ios_base::in|ios_base::out);
    void close();

  private:
    basic_filebuf<charT,traits> sb; // exposition only
  };

  template <class charT, class traits>
  void swap(basic_fstream<charT, traits>& x,
            basic_fstream<charT, traits>& y);
}
```

1 The class template `basic_fstream<charT,traits>` supports reading and writing from named files. It uses a `basic_filebuf<charT,traits>` object to control the associated sequences. For the sake of exposition, the maintained data is presented here as:

(1.1) — `sb`, the `basic_filebuf` object.

27.308.0.2 basic_fstream constructors [fstream.cons]

`basic_fstream();`

1 *Effects:* Constructs an object of class `basic_fstream<charT,traits>`, initializing the base class with `basic_iostream(&sb)` and initializing `sb` with `basic_filebuf<charT,traits>()`.

```
explicit basic_fstream(const char* s,
    ios_base::openmode mode = ios_base::in|ios_base::out);
```

2 *Effects:* Constructs an object of class `basic_fstream<charT, traits>`, initializing the base class with `basic_iostream(&sb)` and initializing `sb` with `basic_filebuf<charT, traits>()`. Then calls `rdbuf()->open(s, mode)`. If that function returns a null pointer, calls `setstate(failbit)`.

```
explicit basic_fstream(const string& s,
    ios_base::openmode mode = ios_base::in|ios_base::out);
```

3 *Effects:* the same as `basic_fstream(s.c_str(), mode);`

```
basic_fstream(basic_fstream&& rhs);
```

4 *Effects:* Move constructs from the rvalue `rhs`. This is accomplished by move constructing the base class, and the contained `basic_filebuf`. Next `basic_istream<charT,traits>::set_rdbuf(&sb)` is called to install the contained `basic_filebuf`.

27.308.0.3 Assign and swap [fstream.assign]

```
basic_fstream& operator=(basic_fstream&& rhs);
```

1 *Effects:* Move assigns the base and members of `*this` from the base and corresponding members of `rhs`.

2 *Returns:* `*this`.

```
void swap(basic_fstream& rhs);
```

3 *Effects:* Exchanges the state of `*this` and `rhs` by calling `basic_iostream<charT,traits>::swap(rhs)` and `sb.swap(rhs.sb)`.

```
template <class charT, class traits>
void swap(basic_fstream<charT, traits>& x,
          basic_fstream<charT, traits>& y);
```

4 *Effects:* `x.swap(y)`.

27.308.0.4 Member functions [fstream.members]

```
basic_filebuf<charT,traits>* rdbuf() const;
```

1 *Returns:* `const_cast<basic_filebuf<charT,traits>*>(&sb)`.

```
bool is_open() const;
```

2 *Returns:* `rdbuf()->is_open()`.

```
void open(const char* s,
    ios_base::openmode mode = ios_base::in|ios_base::out);
```

1 *Effects:* Calls `rdbuf()->open(s,mode)`. If that function does not return a null pointer calls `clear()`, otherwise calls `setstate(failbit)`, (which may throw `ios_base::failure`) (27.84.0.1).

```
void open(const string& s,
    ios_base::openmode mode = ios_base::in|ios_base::out);
```

1 *Effects:* calls `open(s.c_str(), mode);`

```
void close();
```

2 *Effects:* Calls `rdbuf()->close()` and, if that function returns returns a null pointer, calls `setstate(failbit)` (27.84.0.1) (which may throw `ios_base::failure`).

27.310.1 C library files [c.files]

1 Table 134 describes header `<cstdio>`. [*Note:* C++ does not define the function `gets`. — *end note*]

Table 134 — Header `<cstdio>` synopsis

Type	Name(s)				
Macros:					
BUFSIZ	FOPEN_MAX	SEEK_CUR	TMP_MAX	_IONBF	stdout
EOF	L_tmpnam	SEEK_END	_IOFBF	stderr	
FILENAME_MAX	NULL <cstdio>	SEEK_SET	_IOLBF	stdin	
Types:	FILE	fpos_t	size_t <cstdio>		
Functions:					
clearerr	fopen	fsetpos	putchar	snprintf	vscanf
fclose	fprintf	ftell	puts	sprintf	vsnprintf
feof	fputc	fwrite	remove	sscanf	vsprintf
ferror	fputs	getc	rename	tmpfile	vsscanf
fflush	fread	getchar	rewind	tmpnam	
fgetc	freopen	perror	scanf	ungetc	
fgetpos	fscanf	printf	setbuf	vfprintf	
fgets	fseek	putc	setvbuf	vprintf	

2 Calls to the function `tmpnam` with an argument of `NULL` may introduce a data race (17.607.0.1) with other calls to `tmpnam` with an argument of `NULL`.

See also: ISO C 7.9, Amendment 1 4.6.2.

3 Table 135 describes header `<cinttypes>`. [*Note:* The macros defined by `<cinttypes>` are provided unconditionally. In particular, the symbol `__STDC_FORMAT_MACROS`, mentioned in footnote 182 of the C standard, plays no role in C++. — *end note*]

Table 135 — Header `<cinttypes>` synopsis

Type	Name(s)		
Macros:			
PRI{d i o u x X}[FAST LEAST]{8 16 32 64}			
PRI{d i o u x X}{MAX PTR}			
SCN{d i o u x X}[FAST LEAST]{8 16 32 64}			
SCN{d i o u x X}{MAX PTR}			
Types:	imaxdiv_t		
Functions:			
abs	imaxabs	strtoimax	wcstoimax
div	imaxdiv	strtoumax	wcstoumax

4

The contents of header `<cinttypes>` are the same as the Standard C Library header `<inttypes.h>`, with the following changes:

— the header `<cinttypes>` includes the header `<cstdint>` instead of `<stdint.h>`, and

— if and only if the type `intmax_t` designates an extended integer type (3.3151.1), the following function signatures are added:

```
intmax_t abs(intmax_t);
imaxdiv_t div(intmax_t, intmax_t);
```

which shall have the same semantics as the function signatures `intmax_t imaxabs(intmax_t)` and `imaxdiv_t imaxdiv(intmax_t, intmax_t)`, respectively.

28 Regular expressions library [re]

28.1 General [re.general]

1 This Clause describes components that C++ programs may use to perform operations involving regular expression matching and searching.

2 The following subclauses describe a basic regular expression class template and its traits that can handle char-like template arguments, two specializations of this class template that handle sequences of `char` and `wchar_t`, a class template that holds the result of a regular expression match, a series of algorithms that allow a character sequence to be operated upon by a regular expression, and two iterator types for enumerating regular expression matches, as described in Table 136.

Table 136 — Regular expressions library summary

Subclause		Header(s)
28.2	Definitions	
28.3	Requirements	
28.8	Constants	
28.24	Exception type	
28.26	Traits	
28.28	Regular expression template	`<regex>`
28.30	Submatches	
28.31	Match results	
28.33	Algorithms	
28.42	Iterators	
28.45	Grammar	

28.2 Definitions [re.def]

1 The following definitions shall apply to this Clause:

28.2.1 [defns.regex.collating.element]

collating element

a sequence of one or more characters within the current locale that collate as if they were a single character.

28.2.2 [defns.regex.finite.state.machine]

finite state machine

an unspecified data structure that is used to represent a regular expression, and which permits efficient matches against the regular expression to be obtained.

28.2.3 [defns.regex.format.specifier]

format specifier

a sequence of one or more characters that is to be replaced with some part of a regular expression match.

28.2.4 [defns.regex.matched]

matched

a sequence of zero or more characters is matched by a regular expression when the characters in the sequence correspond to a sequence of characters defined by the pattern.

28.2.5 **[defns.regex.primary.equivalence.class]**

primary equivalence class

a set of one or more characters which share the same primary sort key: that is the sort key weighting that depends only upon character shape, and not accents, case, or locale specific tailorings.

28.2.6 **[defns.regex.regular.expression]**

regular expression

a pattern that selects specific strings from a set of character strings.

28.2.7 **[defns.regex.subexpression]**

sub-expression

a subset of a regular expression that has been marked by parenthesis.

28.3 Requirements [re.req]

1 This subclause defines requirements on classes representing regular expression traits. [*Note:* The class template `regex_traits`, defined in Clause 28.26, satisfies these requirements. — *end note*]

2 The class template `basic_regex`, defined in Clause 28.28, needs a set of related types and functions to complete the definition of its semantics. These types and functions are provided as a set of member typedefs and functions in the template parameter `traits` used by the `basic_regex` class template. This subclause defines the semantics of these members.

3 To specialize class template `basic_regex` for a character container `CharT` and its related regular expression traits class `Traits`, use `basic_regex<CharT, Traits>`.

4

In Table 137 `X` denotes a traits class defining types and functions for the character container type `charT`; `u` is an object of type `X`; `v` is an object of type `const X`; `p` is a value of type `const charT*`; `I1` and `I2` are input iterators (24.100.1); `F1` and `F2` are forward iterators (24.122.1); `c` is a value of type `const charT`; `s` is an object of type `X::string_type`; `cs` is an object of type `const X::string_type`; `b` is a value of type `bool`; `I` is a value of type `int`; `cl` is an object of type `X::char_class_type`, and `loc` is an object of type `X::locale_type`.

Table 137 — Regular expression traits class requirements

Expression	Return type	Assertion/note pre-/post-condition
`X::char_type`	`charT`	The character container type used in the implementation of class template `basic_regex`.
`X::string_type`	`std::basic_string<charT>`	
`X::locale_type`	A copy constructible type	A type that represents the locale used by the traits class.
`X::char_class_type`	A bitmask type (17.20.0.0.1).	A bitmask type representing a particular character classification.
`X::length(p)`	`std::size_t`	Yields the smallest `i` such that `p[i] == 0`. Complexity is linear in `i` .

Table 137 — Regular expression traits class requirements (continued)

Expression	Return type	Assertion/note pre-/post-condition
`v.translate(c)`	`X::char_type`	Returns a character such that for any character `d` that is to be considered equivalent to `c` then `v.translate(c) == v.translate(d)`.
`v.translate_nocase(c)`	`X::char_type`	For all characters `C` that are to be considered equivalent to `c` when comparisons are to be performed without regard to case, then `v.translate_nocase(c) == v.translate_nocase(C)`.
`v.transform(F1, F2)`	`X::string_type`	Returns a sort key for the character sequence designated by the iterator range `[F1,F2)` such that if the character sequence `[G1,G2)` sorts before the character sequence `[H1,H2)` then `v.transform(G1, G2) < v.transform(H1, H2)`.
`v.transform_primary(F1, F2)`	`X::string_type`	Returns a sort key for the character sequence designated by the iterator range `[F1,F2)` such that if the character sequence `[G1,G2)` sorts before the character sequence `[H1,H2)` when character case is not considered then `v.transform_primary(G1, G2) < v.transform_primary(H1, H2)`.
`v.lookup_collatename(F1, F2)`	`X::string_type`	Returns a sequence of characters that represents the collating element consisting of the character sequence designated by the iterator range `[F1,F2)`. Returns an empty string if the character sequence is not a valid collating element.
`v.lookup_classname(F1, F2, b)`	`X::char_class_type`	Converts the character sequence designated by the iterator range `[F1,F2)` into a value of a bitmask type that can subsequently be passed to `isctype`. Values returned from `lookup_classname` can be bitwise or'ed together; the resulting value represents membership in either of the corresponding character classes. If `b` is true, the returned bitmask is suitable for matching characters without regard to their case. Returns 0 if the character sequence is not the name of a character class recognized by `X`. The value returned shall be independent of the case of the characters in the sequence.
`v.isctype(c, cl)`	`bool`	Returns `true` if character `c` is a member of one of the character classes designated by `cl`, `false` otherwise.

Table 137 — Regular expression traits class requirements (continued)

Expression	Return type	Assertion/note pre-/post-condition
`v.value(c, I)`	`int`	Returns the value represented by the digit c in base I if the character c is a valid digit in base I; otherwise returns `-1`. [*Note:* The value of I will only be 8, 10, or 16. — *end note*]
`u.imbue(loc)`	`X::locale_type`	Imbues `u` with the locale `loc` and returns the previous locale used by `u` if any.
`v.getloc()`	`X::locale_type`	Returns the current locale used by `v`, if any.

1 [*Note:* Class template `regex_traits` satisfies the requirements for a regular expression traits class when it is specialized for `char` or `wchar_t`. This class template is described in the header `<regex>`, and is described in Clause 28.26. — *end note*]

28.7 Header `<regex>` synopsis [re.syn]

```
#include <initializer_list>

namespace std {

  // 28.8, regex constants:
  namespace regex_constants {
    enum error_type;
  } // namespace regex_constants

  // 28.24, class regex_error:
  class regex_error;

  // 28.26, class template regex_traits:
  template <class charT> struct regex_traits;

  // 28.28, class template basic_regex:
  template <class charT, class traits = regex_traits<charT> > class basic_regex;

  typedef basic_regex<char>    regex;
  typedef basic_regex<wchar_t> wregex;

  // 28.29.6, basic_regex swap:
  template <class charT, class traits>
    void swap(basic_regex<charT, traits>& e1, basic_regex<charT, traits>& e2);

  // 28.30, class template sub_match:
  template <class BidirectionalIterator>
    class sub_match;

  typedef sub_match<const char*>             csub_match;
  typedef sub_match<const wchar_t*>          wcsub_match;
  typedef sub_match<string::const_iterator>  ssub_match;
  typedef sub_match<wstring::const_iterator> wssub_match;

  // 28.30.2, sub_match non-member operators:
```

```
template <class BiIter>
  bool operator==(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator!=(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator<(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator<=(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator>=(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator>(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);


template <class BiIter, class ST, class SA>
  bool operator==(
    const basic_string<
      typename iterator_traits<BiIter>::value_type, ST, SA>& lhs,
    const sub_match<BiIter>& rhs);
template <class BiIter, class ST, class SA>
  bool operator!=(
    const basic_string<
      typename iterator_traits<BiIter>::value_type, ST, SA>& lhs,
    const sub_match<BiIter>& rhs);
template <class BiIter, class ST, class SA>
  bool operator<(
    const basic_string<
      typename iterator_traits<BiIter>::value_type, ST, SA>& lhs,
    const sub_match<BiIter>& rhs);
template <class BiIter, class ST, class SA>
  bool operator>(
    const basic_string<
      typename iterator_traits<BiIter>::value_type, ST, SA>& lhs,
    const sub_match<BiIter>& rhs);
template <class BiIter, class ST, class SA>
  bool operator>=(
    const basic_string<
      typename iterator_traits<BiIter>::value_type, ST, SA>& lhs,
    const sub_match<BiIter>& rhs);
template <class BiIter, class ST, class SA>
  bool operator<=(
    const basic_string<
      typename iterator_traits<BiIter>::value_type, ST, SA>& lhs,
    const sub_match<BiIter>& rhs);

template <class BiIter, class ST, class SA>
  bool operator==(
    const sub_match<BiIter>& lhs,
    const basic_string<
      typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);
template <class BiIter, class ST, class SA>
  bool operator!=(
    const sub_match<BiIter>& lhs,
    const basic_string<
      typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);
```

```
template <class BiIter, class ST, class SA>
  bool operator<(
    const sub_match<BiIter>& lhs,
    const basic_string<
      typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);
template <class BiIter, class ST, class SA>
  bool operator>(
    const sub_match<BiIter>& lhs,
    const basic_string<
      typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);
template <class BiIter, class ST, class SA>
  bool operator>=(
    const sub_match<BiIter>& lhs,
    const basic_string<
      typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);
template <class BiIter, class ST, class SA>
  bool operator<=(
    const sub_match<BiIter>& lhs,
    const basic_string<
      typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);

template <class BiIter>
  bool operator==(typename iterator_traits<BiIter>::value_type const* lhs,
                  const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator!=(typename iterator_traits<BiIter>::value_type const* lhs,
                  const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator<(typename iterator_traits<BiIter>::value_type const* lhs,
                 const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator>(typename iterator_traits<BiIter>::value_type const* lhs,
                 const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator>=(typename iterator_traits<BiIter>::value_type const* lhs,
                  const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator<=(typename iterator_traits<BiIter>::value_type const* lhs,
                  const sub_match<BiIter>& rhs);

template <class BiIter>
  bool operator==(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const* rhs);
template <class BiIter>
  bool operator!=(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const* rhs);
template <class BiIter>
  bool operator<(const sub_match<BiIter>& lhs,
                 typename iterator_traits<BiIter>::value_type const* rhs);
template <class BiIter>
  bool operator>(const sub_match<BiIter>& lhs,
                 typename iterator_traits<BiIter>::value_type const* rhs);
template <class BiIter>
  bool operator>=(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const* rhs);
```

```
template <class BiIter>
  bool operator<=(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const* rhs);

template <class BiIter>
  bool operator==(typename iterator_traits<BiIter>::value_type const& lhs,
                  const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator!=(typename iterator_traits<BiIter>::value_type const& lhs,
                  const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator<(typename iterator_traits<BiIter>::value_type const& lhs,
                 const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator>(typename iterator_traits<BiIter>::value_type const& lhs,
                 const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator>=(typename iterator_traits<BiIter>::value_type const& lhs,
                  const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator<=(typename iterator_traits<BiIter>::value_type const& lhs,
                  const sub_match<BiIter>& rhs);

template <class BiIter>
  bool operator==(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const& rhs);
template <class BiIter>
  bool operator!=(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const& rhs);
template <class BiIter>
  bool operator<(const sub_match<BiIter>& lhs,
                 typename iterator_traits<BiIter>::value_type const& rhs);
template <class BiIter>
  bool operator>(const sub_match<BiIter>& lhs,
                 typename iterator_traits<BiIter>::value_type const& rhs);
template <class BiIter>
  bool operator>=(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const& rhs);
template <class BiIter>
  bool operator<=(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const& rhs);

template <class charT, class ST, class BiIter>
  basic_ostream<charT, ST>&
  operator<<(basic_ostream<charT, ST>& os, const sub_match<BiIter>& m);

// 28.31, class template match_results:
template <class BidirectionalIterator,
          class Allocator = allocator<sub_match<BidirectionalIterator> > >
  class match_results;

typedef match_results<const char*>             cmatch;
typedef match_results<const wchar_t*>          wcmatch;
typedef match_results<string::const_iterator>  smatch;
typedef match_results<wstring::const_iterator> wsmatch;
```

```
// match_results comparisons
template <class BidirectionalIterator, class Allocator>
  bool operator== (const match_results<BidirectionalIterator, Allocator>& m1,
                   const match_results<BidirectionalIterator, Allocator>& m2);
template <class BidirectionalIterator, class Allocator>
  bool operator!= (const match_results<BidirectionalIterator, Allocator>& m1,
                   const match_results<BidirectionalIterator, Allocator>& m2);

// 28.32.6, match_results swap:
template <class BidirectionalIterator, class Allocator>
  void swap(match_results<BidirectionalIterator, Allocator>& m1,
            match_results<BidirectionalIterator, Allocator>& m2);

// 28.35.1, function template regex_match:
template <class BidirectionalIterator, class Allocator,
    class charT, class traits>
  bool regex_match(BidirectionalIterator first, BidirectionalIterator last,
                   match_results<BidirectionalIterator, Allocator>& m,
                   const basic_regex<charT, traits>& e,
                   regex_constants::match_flag_type flags =
                     regex_constants::match_default);
template <class BidirectionalIterator, class charT, class traits>
bool regex_match(BidirectionalIterator first, BidirectionalIterator last,
                 const basic_regex<charT, traits>& e,
                 regex_constants::match_flag_type flags =
                   regex_constants::match_default);
template <class charT, class Allocator, class traits>
  bool regex_match(const charT* str, match_results<const charT*, Allocator>& m,
                   const basic_regex<charT, traits>& e,
                   regex_constants::match_flag_type flags =
                     regex_constants::match_default);
template <class ST, class SA, class Allocator, class charT, class traits>
  bool regex_match(const basic_string<charT, ST, SA>& s,
                   match_results<
                     typename basic_string<charT, ST, SA>::const_iterator,
                     Allocator>& m,
                   const basic_regex<charT, traits>& e,
                   regex_constants::match_flag_type flags =
                     regex_constants::match_default);
template <class ST, class SA, class Allocator, class charT, class traits>
  bool regex_match(const basic_string<charT, ST, SA>&&,
                   match_results<
                     typename basic_string<charT, ST, SA>::const_iterator,
                     Allocator>&,
                   const basic_regex<charT, traits>&,
                   regex_constants::match_flag_type =
                     regex_constants::match_default) = delete;
template <class charT, class traits>
  bool regex_match(const charT* str,
                   const basic_regex<charT, traits>& e,
                   regex_constants::match_flag_type flags =
                     regex_constants::match_default);
template <class ST, class SA, class charT, class traits>
  bool regex_match(const basic_string<charT, ST, SA>& s,
```

```
                    const basic_regex<charT, traits>& e,
                    regex_constants::match_flag_type flags =
                      regex_constants::match_default);

// 28.37.1, function template regex_search:
template <class BidirectionalIterator, class Allocator,
    class charT, class traits>
  bool regex_search(BidirectionalIterator first, BidirectionalIterator last,
                    match_results<BidirectionalIterator, Allocator>& m,
                    const basic_regex<charT, traits>& e,
                    regex_constants::match_flag_type flags =
                      regex_constants::match_default);
template <class BidirectionalIterator, class charT, class traits>
  bool regex_search(BidirectionalIterator first, BidirectionalIterator last,
                    const basic_regex<charT, traits>& e,
                    regex_constants::match_flag_type flags =
                      regex_constants::match_default);
template <class charT, class Allocator, class traits>
  bool regex_search(const charT* str,
                    match_results<const charT*, Allocator>& m,
                    const basic_regex<charT, traits>& e,
                    regex_constants::match_flag_type flags =
                      regex_constants::match_default);
template <class charT, class traits>
  bool regex_search(const charT* str,
                    const basic_regex<charT, traits>& e,
                    regex_constants::match_flag_type flags =
                      regex_constants::match_default);
template <class ST, class SA, class charT, class traits>
  bool regex_search(const basic_string<charT, ST, SA>& s,
                    const basic_regex<charT, traits>& e,
                    regex_constants::match_flag_type flags =
                      regex_constants::match_default);
template <class ST, class SA, class Allocator, class charT, class traits>
  bool regex_search(const basic_string<charT, ST, SA>& s,
                    match_results<
                      typename basic_string<charT, ST, SA>::const_iterator,
                      Allocator>& m,
                    const basic_regex<charT, traits>& e,
                    regex_constants::match_flag_type flags =
                      regex_constants::match_default);
template <class ST, class SA, class Allocator, class charT, class traits>
  bool regex_search(const basic_string<charT, ST, SA>&&,
                    match_results<
                      typename basic_string<charT, ST, SA>::const_iterator,
                      Allocator>&,
                    const basic_regex<charT, traits>&,
                    regex_constants::match_flag_type =
                      regex_constants::match_default) = delete;

// 28.41.1, function template regex_replace:
template <class OutputIterator, class BidirectionalIterator,
    class traits, class charT, class ST, class SA>
  OutputIterator
  regex_replace(OutputIterator out,
```

```
                BidirectionalIterator first, BidirectionalIterator last,
                const basic_regex<charT, traits>& e,
                const basic_string<charT, ST, SA>& fmt,
                regex_constants::match_flag_type flags =
                  regex_constants::match_default);
template <class OutputIterator, class BidirectionalIterator,
    class traits, class charT>
  OutputIterator
  regex_replace(OutputIterator out,
                BidirectionalIterator first, BidirectionalIterator last,
                const basic_regex<charT, traits>& e,
                const charT* fmt,
                regex_constants::match_flag_type flags =
                  regex_constants::match_default);
template <class traits, class charT, class ST, class SA,
    class FST, class FSA>
  basic_string<charT, ST, SA>
  regex_replace(const basic_string<charT, ST, SA>& s,
                const basic_regex<charT, traits>& e,
                const basic_string<charT, FST, FSA>& fmt,
                regex_constants::match_flag_type flags =
                  regex_constants::match_default);
template <class traits, class charT, class ST, class SA>
  basic_string<charT, ST, SA>
  regex_replace(const basic_string<charT, ST, SA>& s,
                const basic_regex<charT, traits>& e,
                const charT* fmt,
                regex_constants::match_flag_type flags =
                  regex_constants::match_default);
template <class traits, class charT, class ST, class SA>
  basic_string<charT>
  regex_replace(const charT* s,
                const basic_regex<charT, traits>& e,
                const basic_string<charT, ST, SA>& fmt,
                regex_constants::match_flag_type flags =
                  regex_constants::match_default);
template <class traits, class charT>
  basic_string<charT>
  regex_replace(const charT* s,
                const basic_regex<charT, traits>& e,
                const charT* fmt,
                regex_constants::match_flag_type flags =
                  regex_constants::match_default);

// 28.42.1, class template regex_iterator:
template <class BidirectionalIterator,
          class charT = typename iterator_traits<
            BidirectionalIterator>::value_type,
          class traits = regex_traits<charT> >
  class regex_iterator;

typedef regex_iterator<const char*>             cregex_iterator;
typedef regex_iterator<const wchar_t*>          wcregex_iterator;
typedef regex_iterator<string::const_iterator>  sregex_iterator;
typedef regex_iterator<wstring::const_iterator> wsregex_iterator;
```

```
  // 28.43.1, class template regex_token_iterator:
  template <class BidirectionalIterator,
            class charT = typename iterator_traits<
              BidirectionalIterator>::value_type,
            class traits = regex_traits<charT> >
    class regex_token_iterator;

  typedef regex_token_iterator<const char*>             cregex_token_iterator;
  typedef regex_token_iterator<const wchar_t*>          wcregex_token_iterator;
  typedef regex_token_iterator<string::const_iterator>  sregex_token_iterator;
  typedef regex_token_iterator<wstring::const_iterator> wsregex_token_iterator;
}
```

28.8 Namespace `std::regex_constants` [re.const]

1 The namespace `std::regex_constants` holds symbolic constants used by the regular expression library. This namespace provides three types, `syntax_option_type`, `match_flag_type`, and `error_type`, along with several constants of these types.

28.8.1 Bitmask type `syntax_option_type` [re.synopt]

```
namespace std {
  namespace regex_constants {
    typedef T1 syntax_option_type;
    constexpr syntax_option_type icase = unspecified ;
    constexpr syntax_option_type nosubs = unspecified ;
    constexpr syntax_option_type optimize = unspecified ;
    constexpr syntax_option_type collate = unspecified ;
    constexpr syntax_option_type ECMAScript = unspecified ;
    constexpr syntax_option_type basic = unspecified ;
    constexpr syntax_option_type extended = unspecified ;
    constexpr syntax_option_type awk = unspecified ;
    constexpr syntax_option_type grep = unspecified ;
    constexpr syntax_option_type egrep = unspecified ;
  }
}
```

1

The type `syntax_option_type` is an implementation-defined bitmask type (17.20.0.0.1). Setting its elements has the effects listed in table 138. A valid value of type `syntax_option_type` shall have at most one of the grammar elements `ECMAScript`, `basic`, `extended`, `awk`, `grep`, `egrep`, set. If no grammar element is set, the default grammar is `ECMAScript`.

28.10.1 Bitmask type `regex_constants::match_flag_type` [re.matchflag]

```
namespace std {
  namespace regex_constants{
    typedef T2 match_flag_type;
    constexpr match_flag_type match_default = {};
    constexpr match_flag_type match_not_bol = unspecified ;
    constexpr match_flag_type match_not_eol = unspecified ;
    constexpr match_flag_type match_not_bow = unspecified ;
    constexpr match_flag_type match_not_eow = unspecified ;
    constexpr match_flag_type match_any = unspecified ;
    constexpr match_flag_type match_not_null = unspecified ;
```

Table 138 — `syntax_option_type` effects

Element	Effect(s) if set
`icase`	Specifies that matching of regular expressions against a character container sequence shall be performed without regard to case.
`nosubs`	Specifies that no sub-expressions shall be considered to be marked, so that when a regular expression is matched against a character container sequence, no sub-expression matches shall be stored in the supplied `match_results` structure.
`optimize`	Specifies that the regular expression engine should pay more attention to the speed with which regular expressions are matched, and less to the speed with which regular expression objects are constructed. Otherwise it has no detectable effect on the program output.
`collate`	Specifies that character ranges of the form `"[a-b]"` shall be locale sensitive.
`ECMAScript`	Specifies that the grammar recognized by the regular expression engine shall be that used by ECMAScript in ECMA-262, as modified in 28.45.
`basic`	Specifies that the grammar recognized by the regular expression engine shall be that used by basic regular expressions in POSIX, Base Definitions and Headers, Section 9, Regular Expressions.
`extended`	Specifies that the grammar recognized by the regular expression engine shall be that used by extended regular expressions in POSIX, Base Definitions and Headers, Section 9, Regular Expressions.
`awk`	Specifies that the grammar recognized by the regular expression engine shall be that used by the utility awk in POSIX.
`grep`	Specifies that the grammar recognized by the regular expression engine shall be that used by the utility grep in POSIX.
`egrep`	Specifies that the grammar recognized by the regular expression engine shall be that used by the utility grep when given the -E option in POSIX.

```
    constexpr match_flag_type match_continuous = unspecified ;
    constexpr match_flag_type match_prev_avail = unspecified ;
    constexpr match_flag_type format_default = {};
    constexpr match_flag_type format_sed = unspecified ;
    constexpr match_flag_type format_no_copy = unspecified ;
    constexpr match_flag_type format_first_only = unspecified ;
  }
}
```

1

The type `regex_constants::match_flag_type` is an implementation-defined bitmask type (17.20.0.0.1). The constants of that type, except for `match_default` and `format_default`, are bitmask elements. The `match_default` and `format_default` constants are empty bitmasks. Matching a regular expression against a sequence of characters `[first,last)` proceeds according to the rules of the grammar specified for the regular expression object, modified according to the effects listed in Table 139 for any bitmask elements set.

Table 139 — `regex_constants::match_flag_type` effects when obtaining a match against a character container sequence `[first, last)`.

Element	Effect(s) if set
`match_not_bol`	The first character in the sequence `[first,last)` shall be treated as though it is not at the beginning of a line, so the character `^` in the regular expression shall not match `[first,first)`.
`match_not_eol`	The last character in the sequence `[first,last)` shall be treated as though it is not at the end of a line, so the character `"$"` in the regular expression shall not match `[last,last)`.
`match_not_bow`	The expression `"\\b"` shall not match the sub-sequence `[first,first)`.
`match_not_eow`	The expression `"\\b"` shall not match the sub-sequence `[last,last)`.
`match_any`	If more than one match is possible then any match is an acceptable result.
`match_not_null`	The expression shall not match an empty sequence.
`match_continuous`	The expression shall only match a sub-sequence that begins at `first`.
`match_prev_avail`	`--first` is a valid iterator position. When this flag is set the flags match_not_bol and match_not_bow shall be ignored by the regular expression algorithms 28.33 and iterators 28.42.
`format_default`	When a regular expression match is to be replaced by a new string, the new string shall be constructed using the rules used by the ECMAScript replace function in ECMA-262, part 15.5.4.11 String.prototype.replace. In addition, during search and replace operations all non-overlapping occurrences of the regular expression shall be located and replaced, and sections of the input that did not match the expression shall be copied unchanged to the output string.
`format_sed`	When a regular expression match is to be replaced by a new string, the new string shall be constructed using the rules used by the sed utility in POSIX.
`format_no_copy`	During a search and replace operation, sections of the character container sequence being searched that do not match the regular expression shall not be copied to the output string.
`format_first_only`	When specified during a search and replace operation, only the first occurrence of the regular expression shall be replaced.

28.19.1 Implementation-defined `error_type` [re.err]

```
namespace std {
  namespace regex_constants {
    typedef T3 error_type;
    constexpr error_type error_collate = unspecified ;
    constexpr error_type error_ctype = unspecified ;
    constexpr error_type error_escape = unspecified ;
    constexpr error_type error_backref = unspecified ;
    constexpr error_type error_brack = unspecified ;
    constexpr error_type error_paren = unspecified ;
    constexpr error_type error_brace = unspecified ;
    constexpr error_type error_badbrace = unspecified ;
    constexpr error_type error_range = unspecified ;
    constexpr error_type error_space = unspecified ;
    constexpr error_type error_badrepeat = unspecified ;
    constexpr error_type error_complexity = unspecified ;
```

```
    constexpr error_type error_stack = unspecified ;
  }
}
```

1

The type `error_type` is an implementation-defined enumerated type (17.17.1.1.2). Values of type `error_type` represent the error conditions described in Table 140:

Table 140 — `error_type` values in the C locale

Value	Error condition
`error_collate`	The expression contained an invalid collating element name.
`error_ctype`	The expression contained an invalid character class name.
`error_escape`	The expression contained an invalid escaped character, or a trailing escape.
`error_backref`	The expression contained an invalid back reference.
`error_brack`	The expression contained mismatched `[` and `]`.
`error_paren`	The expression contained mismatched `(` and `)`.
`error_brace`	The expression contained mismatched `{` and `}`
`error_badbrace`	The expression contained an invalid range in a `{}` expression.
`error_range`	The expression contained an invalid character range, such as `[b-a]` in most encodings.
`error_space`	There was insufficient memory to convert the expression into a finite state machine.
`error_badrepeat`	One of `*?+{` was not preceded by a valid regular expression.
`error_complexity`	The complexity of an attempted match against a regular expression exceeded a pre-set level.
`error_stack`	There was insufficient memory to determine whether the regular expression could match the specified character sequence.

28.24 Class `regex_error` [re.badexp]

```
class regex_error : public std::runtime_error {
  public:
    explicit regex_error(regex_constants::error_type ecode);
    regex_constants::error_type code() const;
};
```

1

The class `regex_error` defines the type of objects thrown as exceptions to report errors from the regular expression library.

`regex_error(regex_constants::error_type ecode);`

1 *Effects:* Constructs an object of class `regex_error`.

2 *Postcondition:*: `ecode == code()`

`regex_constants::error_type code() const;`

3 *Returns:* The error code that was passed to the constructor.

28.26 Class template `regex_traits` [re.traits]

```
namespace std {
  template <class charT>
  struct regex_traits {
  public:
     typedef charT                        char_type;
     typedef std::basic_string<char_type> string_type;
     typedef std::locale                  locale_type;
     typedef bitmask_type                 char_class_type;

     regex_traits();
     static std::size_t length(const char_type* p);
     charT translate(charT c) const;
     charT translate_nocase(charT c) const;
     template <class ForwardIterator>
       string_type transform(ForwardIterator first, ForwardIterator last) const;
     template <class ForwardIterator>
       string_type transform_primary(
         ForwardIterator first, ForwardIterator last) const;
     template <class ForwardIterator>
       string_type lookup_collatename(
         ForwardIterator first, ForwardIterator last) const;
     template <class ForwardIterator>
       char_class_type lookup_classname(
         ForwardIterator first, ForwardIterator last, bool icase = false) const;
     bool isctype(charT c, char_class_type f) const;
     int value(charT ch, int radix) const;
     locale_type imbue(locale_type l);
     locale_type getloc() const;
  };
}
```

1 The specializations `regex_traits<char>` and `regex_traits<wchar_t>` shall be valid and shall satisfy the requirements for a regular expression traits class (28.3).

`typedef bitmask_type  char_class_type;`

2 The type `char_class_type` is used to represent a character classification and is capable of holding an implementation specific set returned by `lookup_classname`.

`static std::size_t length(const char_type* p);`

3 *Returns:* `char_traits<charT>::length(p);`

`charT translate(charT c) const;`

4 *Returns:* `(c)`.

`charT translate_nocase(charT c) const;`

5 *Returns:* `use_facet<ctype<charT> >(getloc()).tolower(c)`.

```
template <class ForwardIterator>
  string_type transform(ForwardIterator first, ForwardIterator last) const;
```

6 *Effects:*

```
string_type str(first, last);
return use_facet<collate<charT> >(
  getloc()).transform(&*str.begin(), &*str.begin() + str.length());
```

```
template <class ForwardIterator>
  string_type transform_primary(ForwardIterator first, ForwardIterator last) const;
```

7 *Effects:* if `typeid(use_facet<collate<charT> >) == typeid(collate_byname<charT>)` and the form of the sort key returned by `collate_byname<charT> ::transform(first, last)` is known and can be converted into a primary sort key then returns that key, otherwise returns an empty string.

```
template <class ForwardIterator>
  string_type lookup_collatename(ForwardIterator first, ForwardIterator last) const;
```

1 *Returns:* a sequence of one or more characters that represents the collating element consisting of the character sequence designated by the iterator range `[first,last)`. Returns an empty string if the character sequence is not a valid collating element.

```
template <class ForwardIterator>
  char_class_type lookup_classname(
    ForwardIterator first, ForwardIterator last, bool icase = false) const;
```

2 *Returns:* an unspecified value that represents the character classification named by the character sequence designated by the iterator range `[first,last)`. If the parameter `icase` is true then the returned mask identifies the character classification without regard to the case of the characters being matched, otherwise it does honor the case of the characters being matched.[335] The value returned shall be independent of the case of the characters in the character sequence. If the name is not recognized then returns `char_class_type()`.

3 *Remarks:* For `regex_traits<char>`, at least the narrow character names in Table 141 shall be recognized. For `regex_traits<wchar_t>`, at least the wide character names in Table 141 shall be recognized.

```
bool isctype(charT c, char_class_type f) const;
```

4 *Effects:* Determines if the character `c` is a member of the character classification represented by `f`.

5 *Returns:* Given the following function prototype:

```
// for exposition only
template<class C>
  ctype_base::mask convert(typename regex_traits<C>::char_class_type f);
```

that returns a value in which each `ctype_base::mask` value corresponding to a value in `f` named in Table 141 is set, then the result is determined as if by:

```
ctype_base::mask m = convert<charT>(f);
const ctype<charT>& ct = use_facet<ctype<charT>>(getloc());
if (ct.is(m, c)) {
  return true;
} else if (c == ct.widen('_')) {
  charT w[1] = { ct.widen('w') };
  char_class_type x = lookup_classname(w, w+1);

  return (f&x) == x;
} else {
  return false;
}
```

[*Example:*

335) For example, if the parameter `icase` is true then `[[:lower:]]` is the same as `[[:alpha:]]`.

```
regex_traits<char> t;
string d("d");
string u("upper");
regex_traits<char>::char_class_type f;
f = t.lookup_classname(d.begin(), d.end());
f |= t.lookup_classname(u.begin(), u.end());
ctype_base::mask m = convert<char>(f); // m == ctype_base::digit|ctype_base::upper
```

— *end example*] [*Example:*

```
regex_traits<char> t;
string w("w");
regex_traits<char>::char_class_type f;
f = t.lookup_classname(w.begin(), w.end());
t.isctype('A', f); // returns true
t.isctype('_', f); // returns true
t.isctype(' ', f); // returns false
```

— *end example*]

`int value(charT ch, int radix) const;`

6 *Requires:* The value of *radix* shall be 8, 10, or 16.

7 *Returns:* the value represented by the digit *ch* in base *radix* if the character *ch* is a valid digit in base *radix*; otherwise returns -1.

`locale_type imbue(locale_type loc);`

8 *Effects:* Imbues `this` with a copy of the locale `loc`. [*Note:* Calling `imbue` with a different locale than the one currently in use invalidates all cached data held by `*this`. — *end note*]

9 *Returns:* if no locale has been previously imbued then a copy of the global locale in effect at the time of construction of `*this`, otherwise a copy of the last argument passed to `imbue`.

10 *Postcondition:* `getloc() == loc`.

`locale_type getloc() const;`

11 *Returns:* if no locale has been imbued then a copy of the global locale in effect at the time of construction of `*this`, otherwise a copy of the last argument passed to `imbue`.

28.28 Class template `basic_regex` [re.regex]

1 For a char-like type `charT`, specializations of class template `basic_regex` represent regular expressions constructed from character sequences of `charT` characters. In the rest of 28.28, `charT` denotes a given char-like type. Storage for a regular expression is allocated and freed as necessary by the member functions of class `basic_regex`.

2

Objects of type specialization of `basic_regex` are responsible for converting the sequence of `charT` objects to an internal representation. It is not specified what form this representation takes, nor how it is accessed by algorithms that operate on regular expressions. [*Note:* Implementations will typically declare some function templates as friends of `basic_regex` to achieve this — *end note*]

1 The functions described in this Clause report errors by throwing exceptions of type `regex_error`.

Table 141 — Character class names and corresponding `ctype` masks

Narrow character name	Wide character name	Corresponding `ctype_base::mask` value
"alnum"	L"alnum"	ctype_base::alnum
"alpha"	L"alpha"	ctype_base::alpha
"blank"	L"blank"	ctype_base::blank
"cntrl"	L"cntrl"	ctype_base::cntrl
"digit"	L"digit"	ctype_base::digit
"d"	L"d"	ctype_base::digit
"graph"	L"graph"	ctype_base::graph
"lower"	L"lower"	ctype_base::lower
"print"	L"print"	ctype_base::print
"punct"	L"punct"	ctype_base::punct
"space"	L"space"	ctype_base::space
"s"	L"s"	ctype_base::space
"upper"	L"upper"	ctype_base::upper
"w"	L"w"	ctype_base::alnum
"xdigit"	L"xdigit"	ctype_base::xdigit

```
namespace std {
  template <class charT,
            class traits = regex_traits<charT> >
  class basic_regex {
  public:
    // types:
    typedef          charT                                value_type;
    typedef          traits                               traits_type;
    typedef typename traits::string_type                  string_type;
    typedef          regex_constants::syntax_option_type  flag_type;
    typedef typename traits::locale_type                  locale_type;

    // 28.29.1, constants:
    static constexpr regex_constants::syntax_option_type
      icase = regex_constants::icase;
    static constexpr regex_constants::syntax_option_type
      nosubs = regex_constants::nosubs;
    static constexpr regex_constants::syntax_option_type
      optimize = regex_constants::optimize;
    static constexpr regex_constants::syntax_option_type
      collate = regex_constants::collate;
    static constexpr regex_constants::syntax_option_type
      ECMAScript = regex_constants::ECMAScript;
    static constexpr regex_constants::syntax_option_type
      basic = regex_constants::basic;
    static constexpr regex_constants::syntax_option_type
      extended = regex_constants::extended;
    static constexpr regex_constants::syntax_option_type
      awk = regex_constants::awk;
    static constexpr regex_constants::syntax_option_type
      grep = regex_constants::grep;
    static constexpr regex_constants::syntax_option_type
      egrep = regex_constants::egrep;
```

```
    // 28.29.2, construct/copy/destroy:
    basic_regex();
    explicit basic_regex(const charT* p,
      flag_type f = regex_constants::ECMAScript);
    basic_regex(const charT* p, size_t len, flag_type f = regex_constants::ECMAScript);
    basic_regex(const basic_regex&);
    basic_regex(basic_regex&&) noexcept;
    template <class ST, class SA>
      explicit basic_regex(const basic_string<charT, ST, SA>& p,
                           flag_type f = regex_constants::ECMAScript);
    template <class ForwardIterator>
      basic_regex(ForwardIterator first, ForwardIterator last,
                  flag_type f = regex_constants::ECMAScript);
    basic_regex(initializer_list<charT>,
      flag_type = regex_constants::ECMAScript);

    ~basic_regex();

    basic_regex& operator=(const basic_regex&);
    basic_regex& operator=(basic_regex&&) noexcept;
    basic_regex& operator=(const charT* ptr);
    basic_regex& operator=(initializer_list<charT> il);
    template <class ST, class SA>
      basic_regex& operator=(const basic_string<charT, ST, SA>& p);

    // 28.29.3, assign:
    basic_regex& assign(const basic_regex& that);
    basic_regex& assign(basic_regex&& that) noexcept;
    basic_regex& assign(const charT* ptr,
      flag_type f = regex_constants::ECMAScript);
    basic_regex& assign(const charT* p, size_t len, flag_type f);
    template <class string_traits, class A>
      basic_regex& assign(const basic_string<charT, string_traits, A>& s,
                          flag_type f = regex_constants::ECMAScript);
    template <class InputIterator>
      basic_regex& assign(InputIterator first, InputIterator last,
                          flag_type f = regex_constants::ECMAScript);
    basic_regex& assign(initializer_list<charT>,
                        flag_type = regex_constants::ECMAScript);

    // 28.29.4, const operations:
    unsigned mark_count() const;
    flag_type flags() const;

    // 28.29.5, locale:
    locale_type imbue(locale_type loc);
    locale_type getloc() const;

    // 28.29.6, swap:
    void swap(basic_regex&);
  };
}
```

28.29.1 basic_regex constants [re.regex.const]

```
static constexpr regex_constants::syntax_option_type
  icase = regex_constants::icase;
static constexpr regex_constants::syntax_option_type
  nosubs = regex_constants::nosubs;
static constexpr regex_constants::syntax_option_type
  optimize = regex_constants::optimize;
static constexpr regex_constants::syntax_option_type
  collate = regex_constants::collate;
static constexpr regex_constants::syntax_option_type
  ECMAScript = regex_constants::ECMAScript;
static constexpr regex_constants::syntax_option_type
  basic = regex_constants::basic;
static constexpr regex_constants::syntax_option_type
  extended = regex_constants::extended;
static constexpr regex_constants::syntax_option_type
  awk = regex_constants::awk;
static constexpr regex_constants::syntax_option_type
  grep = regex_constants::grep;
static constexpr regex_constants::syntax_option_type
  egrep = regex_constants::egrep;
```

1 The static constant members are provided as synonyms for the constants declared in namespace `regex_constants`.

28.29.2 `basic_regex` constructors [re.regex.construct]

`basic_regex();`

1 *Effects:* Constructs an object of class `basic_regex` that does not match any character sequence.

`explicit basic_regex(const charT* p, flag_type f = regex_constants::ECMAScript);`

2 *Requires:* p shall not be a null pointer.

3 *Throws:* `regex_error` if p is not a valid regular expression.

4 *Effects:* Constructs an object of class `basic_regex`; the object's internal finite state machine is constructed from the regular expression contained in the array of `charT` of length `char_traits<charT>::length(p)` whose first element is designated by p, and interpreted according to the flags f.

5 *Postconditions:* `flags()` returns `f`. `mark_count()` returns the number of marked sub-expressions within the expression.

`basic_regex(const charT* p, size_t len, flag_type f);`

6 *Requires:* p shall not be a null pointer.

7 *Throws:* `regex_error` if p is not a valid regular expression.

8 *Effects:* Constructs an object of class `basic_regex`; the object's internal finite state machine is constructed from the regular expression contained in the sequence of characters `[p,p+len)`, and interpreted according the flags specified in f.

9 *Postconditions:* `flags()` returns `f`. `mark_count()` returns the number of marked sub-expressions within the expression.

`basic_regex(const basic_regex& e);`

10 *Effects:* Constructs an object of class `basic_regex` as a copy of the object `e`.

11 *Postconditions:* `flags()` and `mark_count()` return `e.flags()` and `e.mark_count()`, respectively.

```
basic_regex(basic_regex&& e) noexcept;
```

12 *Effects:* Move constructs an object of class `basic_regex` from `e`.

13 *Postconditions:* `flags()` and `mark_count()` return the values that `e.flags()` and `e.mark_count()`, respectively, had before construction. `e` is in a valid state with unspecified value.

```
template <class ST, class SA>
  explicit basic_regex(const basic_string<charT, ST, SA>& s,
                       flag_type f = regex_constants::ECMAScript);
```

14 *Throws:* `regex_error` if `s` is not a valid regular expression.

15 *Effects:* Constructs an object of class `basic_regex`; the object's internal finite state machine is constructed from the regular expression contained in the string `s`, and interpreted according to the flags specified in `f`.

16 *Postconditions:* `flags()` returns `f`. `mark_count()` returns the number of marked sub-expressions within the expression.

```
template <class ForwardIterator>
  basic_regex(ForwardIterator first, ForwardIterator last,
              flag_type f = regex_constants::ECMAScript);
```

17 *Throws:* `regex_error` if the sequence `[first,last)` is not a valid regular expression.

18 *Effects:* Constructs an object of class `basic_regex`; the object's internal finite state machine is constructed from the regular expression contained in the sequence of characters `[first,last)`, and interpreted according to the flags specified in `f`.

19 *Postconditions:* `flags()` returns `f`. `mark_count()` returns the number of marked sub-expressions within the expression.

```
basic_regex(initializer_list<charT> il,
            flag_type f = regex_constants::ECMAScript);
```

20 *Effects:* Same as `basic_regex(il.begin(), il.end(), f)`.

28.29.3 `basic_regex` assign [re.regex.assign]

```
basic_regex& operator=(const basic_regex& e);
```

1 *Effects:* returns `assign(e)`.

```
basic_regex& operator=(basic_regex&& e) noexcept;
```

2 *Effects:* returns `assign(std::move(e))`.

```
basic_regex& operator=(const charT* ptr);
```

3 *Requires:* `ptr` shall not be a null pointer.

4 *Effects:* returns `assign(ptr)`.

```
basic_regex& operator=(initializer_list<charT> il);
```

5 *Effects:* returns `assign(il.begin(), il.end())`.

```
template <class ST, class SA>
  basic_regex& operator=(const basic_string<charT, ST, SA>& p);
```

6 *Effects:* returns `assign(p)`.

```
basic_regex& assign(const basic_regex& that);
```

7 *Effects:* copies `that` into `*this` and returns `*this`.

8 *Postconditions:* `flags()` and `mark_count()` return `that.flags()` and `that.mark_count()`, respectively.

```
basic_regex& assign(basic_regex&& that) noexcept;
```

9 *Effects:* move assigns from `that` into `*this` and returns `*this`.

10 *Postconditions:* `flags()` and `mark_count()` return the values that `that.flags()` and `that.mark_count()`, respectively, had before assignment. `that` is in a valid state with unspecified value.

```
basic_regex& assign(const charT* ptr, flag_type f = regex_constants::ECMAScript);
```

11 *Returns:* `assign(string_type(ptr), f)`.

```
basic_regex& assign(const charT* ptr, size_t len,
  flag_type f = regex_constants::ECMAScript);
```

12 *Returns:* `assign(string_type(ptr, len), f)`.

```
template <class string_traits, class A>
  basic_regex& assign(const basic_string<charT, string_traits, A>& s,
                      flag_type f = regex_constants::ECMAScript);
```

13 *Throws:* `regex_error` if `s` is not a valid regular expression.

14 *Returns:* `*this`.

15 *Effects:* Assigns the regular expression contained in the string `s`, interpreted according the flags specified in `f`. If an exception is thrown, `*this` is unchanged.

16 *Postconditions:* If no exception is thrown, `flags()` returns `f` and `mark_count()` returns the number of marked sub-expressions within the expression.

```
template <class InputIterator>
  basic_regex& assign(InputIterator first, InputIterator last,
                      flag_type f = regex_constants::ECMAScript);
```

17 *Requires:* The type `InputIterator` shall satisfy the requirements for an Input Iterator (24.100.1).

18 *Returns:* `assign(string_type(first, last), f)`.

```
basic_regex& assign(initializer_list<charT> il,
                    flag_type f = regex_constants::ECMAScript);
```

19 *Effects:* Same as `assign(il.begin(), il.end(), f)`.

20 *Returns:* `*this`.

28.29.4 `basic_regex` constant operations [re.regex.operations]

```
unsigned mark_count() const;
```

1 *Effects:* Returns the number of marked sub-expressions within the regular expression.

```
flag_type flags() const;
```

2 *Effects:* Returns a copy of the regular expression syntax flags that were passed to the object's constructor or to the last call to `assign`.

28.29.5 basic_regex locale [re.regex.locale]

`locale_type imbue(locale_type loc);`

1 *Effects:* Returns the result of `traits_inst.imbue(loc)` where `traits_inst` is a (default initialized) instance of the template type argument `traits` stored within the object. After a call to `imbue` the `basic_regex` object does not match any character sequence.

`locale_type getloc() const;`

2 *Effects:* Returns the result of `traits_inst.getloc()` where `traits_inst` is a (default initialized) instance of the template parameter `traits` stored within the object.

28.29.6 basic_regex swap [re.regex.swap]

`void swap(basic_regex& e);`

1 *Effects:* Swaps the contents of the two regular expressions.

2 *Postcondition:* `*this` contains the regular expression that was in `e`, `e` contains the regular expression that was in `*this`.

3 *Complexity:* Constant time.

28.29.7 basic_regex non-member functions [re.regex.nonmemb]

28.29.7.1 basic_regex non-member swap [re.regex.nmswap]

```
template <class charT, class traits>
  void swap(basic_regex<charT, traits>& lhs, basic_regex<charT, traits>& rhs);
```

1 *Effects:* Calls `lhs.swap(rhs)`.

28.30 Class template sub_match [re.submatch]

1 Class template `sub_match` denotes the sequence of characters matched by a particular marked sub-expression.

```
namespace std {
  template <class BidirectionalIterator>
  class sub_match : public std::pair<BidirectionalIterator, BidirectionalIterator> {
  public:
     typedef typename iterator_traits<BidirectionalIterator>::
       value_type                                          value_type;
     typedef typename iterator_traits<BidirectionalIterator>::
       difference_type                                     difference_type;
     typedef BidirectionalIterator                         iterator;
     typedef basic_string<value_type>                      string_type;

     bool matched;

     constexpr sub_match();

     difference_type length() const;
     operator string_type() const;
     string_type str() const;

     int compare(const sub_match& s) const;
     int compare(const string_type& s) const;
     int compare(const value_type* s) const;
  };
}
```

28.30.1 sub_match members [re.submatch.members]

```
constexpr sub_match();
```

1 *Effects:* Value-initializes the `pair` base class subobject and the member `matched`.

```
difference_type length() const;
```

2 *Returns:* `(matched ? distance(first, second) : 0)`.

```
operator string_type() const;
```

3 *Returns:* `matched ? string_type(first, second) : string_type()`.

```
string_type str() const;
```

4 *Returns:* `matched ? string_type(first, second) : string_type()`.

```
int compare(const sub_match& s) const;
```

5 *Returns:* `str().compare(s.str())`.

```
int compare(const string_type& s) const;
```

6 *Returns:* `str().compare(s)`.

```
int compare(const value_type* s) const;
```

7 *Returns:* `str().compare(s)`.

28.30.2 sub_match non-member operators [re.submatch.op]

```
template <class BiIter>
  bool operator==(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
```

1 *Returns:* `lhs.compare(rhs) == 0`.

```
template <class BiIter>
  bool operator!=(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
```

2 *Returns:* `lhs.compare(rhs) != 0`.

```
template <class BiIter>
  bool operator<(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
```

3 *Returns:* `lhs.compare(rhs) < 0`.

```
template <class BiIter>
  bool operator<=(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
```

4 *Returns:* `lhs.compare(rhs) <= 0`.

```
template <class BiIter>
  bool operator>=(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
```

5 *Returns:* `lhs.compare(rhs) >= 0`.

```
template <class BiIter>
  bool operator>(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
```

6 *Returns:* `lhs.compare(rhs) > 0`.

```
template <class BiIter, class ST, class SA>
  bool operator==(
    const basic_string<
      typename iterator_traits<BiIter>::value_type, ST, SA>& lhs,
    const sub_match<BiIter>& rhs);
```

7 *Returns:* `rhs.compare(lhs.c_str()) == 0`.

```
template <class BiIter, class ST, class SA>
  bool operator!=(
    const basic_string<
      typename iterator_traits<BiIter>::value_type, ST, SA>& lhs,
    const sub_match<BiIter>& rhs);
```

8 *Returns:* `!(lhs == rhs)`.

```
template <class BiIter, class ST, class SA>
  bool operator<(
    const basic_string<
      typename iterator_traits<BiIter>::value_type, ST, SA>& lhs,
    const sub_match<BiIter>& rhs);
```

9 *Returns:* `rhs.compare(lhs.c_str()) > 0`.

```
template <class BiIter, class ST, class SA>
  bool operator>(
    const basic_string<
      typename iterator_traits<BiIter>::value_type, ST, SA>& lhs,
    const sub_match<BiIter>& rhs);
```

10 *Returns:* `rhs < lhs`.

```
template <class BiIter, class ST, class SA>
  bool operator>=(
    const basic_string<
      typename iterator_traits<BiIter>::value_type, ST, SA>& lhs,
    const sub_match<BiIter>& rhs);
```

11 *Returns:* `!(lhs < rhs)`.

```
template <class BiIter, class ST, class SA>
  bool operator<=(
    const basic_string<
      typename iterator_traits<BiIter>::value_type, ST, SA>& lhs,
    const sub_match<BiIter>& rhs);
```

12 *Returns:* `!(rhs < lhs)`.

```
template <class BiIter, class ST, class SA>
  bool operator==(const sub_match<BiIter>& lhs,
                  const basic_string<
                    typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);
```

13 *Returns:* `lhs.compare(rhs.c_str()) == 0`.

```
template <class BiIter, class ST, class SA>
  bool operator!=(const sub_match<BiIter>& lhs,
                  const basic_string<
                    typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);
```

14 *Returns:* `!(lhs == rhs)`.

```
template <class BiIter, class ST, class SA>
  bool operator<(const sub_match<BiIter>& lhs,
                 const basic_string<
                   typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);
```

15 *Returns:* `lhs.compare(rhs.c_str()) < 0`.

```
template <class BiIter, class ST, class SA>
  bool operator>(const sub_match<BiIter>& lhs,
                 const basic_string<
                   typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);
```

16 *Returns:* `rhs < lhs`.

```
template <class BiIter, class ST, class SA>
  bool operator>=(const sub_match<BiIter>& lhs,
                  const basic_string<
                    typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);
```

17 *Returns:* `!(lhs < rhs)`.

```
template <class BiIter, class ST, class SA>
  bool operator<=(const sub_match<BiIter>& lhs,
                  const basic_string<
                    typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);
```

18 *Returns:* `!(rhs < lhs)`.

```
template <class BiIter>
  bool operator==(typename iterator_traits<BiIter>::value_type const* lhs,
                  const sub_match<BiIter>& rhs);
```

19 *Returns:* `rhs.compare(lhs) == 0`.

```
template <class BiIter>
  bool operator!=(typename iterator_traits<BiIter>::value_type const* lhs,
                  const sub_match<BiIter>& rhs);
```

20 *Returns:* `!(lhs == rhs)`.

```
template <class BiIter>
  bool operator<(typename iterator_traits<BiIter>::value_type const* lhs,
                 const sub_match<BiIter>& rhs);
```

21 *Returns:* `rhs.compare(lhs) > 0`.

```
template <class BiIter>
  bool operator>(typename iterator_traits<BiIter>::value_type const* lhs,
                 const sub_match<BiIter>& rhs);
```

22 *Returns:* `rhs < lhs`.

```
template <class BiIter>
  bool operator>=(typename iterator_traits<BiIter>::value_type const* lhs,
                  const sub_match<BiIter>& rhs);
```

23 *Returns:* `!(lhs < rhs)`.

```
template <class BiIter>
  bool operator<=(typename iterator_traits<BiIter>::value_type const* lhs,
                  const sub_match<BiIter>& rhs);
```

24 *Returns:* `!(rhs < lhs)`.

```
template <class BiIter>
  bool operator==(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const* rhs);
```

25 *Returns:* `lhs.compare(rhs) == 0`.

```
template <class BiIter>
  bool operator!=(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const* rhs);
```

26 *Returns:* `!(lhs == rhs)`.

```
template <class BiIter>
  bool operator<(const sub_match<BiIter>& lhs,
                 typename iterator_traits<BiIter>::value_type const* rhs);
```

27 *Returns:* `lhs.compare(rhs) < 0`.

```
template <class BiIter>
  bool operator>(const sub_match<BiIter>& lhs,
                 typename iterator_traits<BiIter>::value_type const* rhs);
```

28 *Returns:* `rhs < lhs`.

```
template <class BiIter>
  bool operator>=(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const* rhs);
```

29 *Returns:* `!(lhs < rhs)`.

```
template <class BiIter>
  bool operator<=(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const* rhs);
```

30 *Returns:* `!(rhs < lhs)`.

```
template <class BiIter>
  bool operator==(typename iterator_traits<BiIter>::value_type const& lhs,
                  const sub_match<BiIter>& rhs);
```

31 *Returns:* `rhs.compare(typename sub_match<BiIter>::string_type(1, lhs)) == 0`.

```
template <class BiIter>
  bool operator!=(typename iterator_traits<BiIter>::value_type const& lhs,
                  const sub_match<BiIter>& rhs);
```

32 *Returns:* `!(lhs == rhs)`.

```
template <class BiIter>
  bool operator<(typename iterator_traits<BiIter>::value_type const& lhs,
                 const sub_match<BiIter>& rhs);
```

33 *Returns:* `rhs.compare(typename sub_match<BiIter>::string_type(1, lhs)) > 0`.

```
template <class BiIter>
  bool operator>(typename iterator_traits<BiIter>::value_type const& lhs,
                 const sub_match<BiIter>& rhs);
```

34 *Returns:* `rhs < lhs`.

```
template <class BiIter>
  bool operator>=(typename iterator_traits<BiIter>::value_type const& lhs,
                  const sub_match<BiIter>& rhs);
```

35 *Returns:* `!(lhs < rhs)`.

```
template <class BiIter>
  bool operator<=(typename iterator_traits<BiIter>::value_type const& lhs,
                  const sub_match<BiIter>& rhs);
```

36 *Returns:* `!(rhs < lhs)`.

```
template <class BiIter>
  bool operator==(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const& rhs);
```

37 *Returns:* `lhs.compare(typename sub_match<BiIter>::string_type(1, rhs)) == 0`.

```
template <class BiIter>
  bool operator!=(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const& rhs);
```

38 *Returns:* `!(lhs == rhs)`.

```
template <class BiIter>
  bool operator<(const sub_match<BiIter>& lhs,
                 typename iterator_traits<BiIter>::value_type const& rhs);
```

39 *Returns:* `lhs.compare(typename sub_match<BiIter>::string_type(1, rhs)) < 0`.

```
template <class BiIter>
  bool operator>(const sub_match<BiIter>& lhs,
                 typename iterator_traits<BiIter>::value_type const& rhs);
```

40 *Returns:* `rhs < lhs`.

```
template <class BiIter>
  bool operator>=(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const& rhs);
```

41 *Returns:* `!(lhs < rhs)`.

```
template <class BiIter>
  bool operator<=(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const& rhs);
```

42 *Returns:* `!(rhs < lhs)`.

```
template <class charT, class ST, class BiIter>
  basic_ostream<charT, ST>&
  operator<<(basic_ostream<charT, ST>& os, const sub_match<BiIter>& m);
```

43 *Returns:* `(os << m.str())`.

28.31 Class template match_results [re.results]

1 Class template `match_results` denotes a collection of character sequences representing the result of a regular expression match. Storage for the collection is allocated and freed as necessary by the member functions of class template `match_results`.

2 The class template `match_results` shall satisfy the requirements of an allocator-aware container and of a sequence container, as specified in 23.224.1, except that only operations defined for const-qualified sequence containers are supported.

3 A default-constructed `match_results` object has no fully established result state. A match result is *ready* when, as a consequence of a completed regular expression match modifying such an object, its result state becomes fully established. The effects of calling most member functions from a `match_results` object that is not ready are undefined.

4 The `sub_match` object stored at index 0 represents sub-expression 0, i.e., the whole match. In this case the `sub_match` member `matched` is always true. The `sub_match` object stored at index `n` denotes what matched the marked sub-expression `n` within the matched expression. If the sub-expression `n` participated in a regular expression match then the `sub_match` member `matched` evaluates to true, and members `first` and `second` denote the range of characters `[first,second)` which formed that match. Otherwise `matched` is false, and members `first` and `second` point to the end of the sequence that was searched. [*Note:* The `sub_match` objects representing different sub-expressions that did not participate in a regular expression match need not be distinct. — *end note*]

```
namespace std {
  template <class BidirectionalIterator,
            class Allocator = allocator<sub_match<BidirectionalIterator>>>
  class match_results {
  public:
     typedef sub_match<BidirectionalIterator>                      value_type;
     typedef const value_type&                                     const_reference;
     typedef value_type&                                           reference;
     typedef implementation-defined                                const_iterator;
     typedef const_iterator                                        iterator;
     typedef typename
      iterator_traits<BidirectionalIterator>::difference_type      difference_type;
     typedef typename allocator_traits<Allocator>::size_type       size_type;
     typedef Allocator                                             allocator_type;
     typedef typename iterator_traits<BidirectionalIterator>::
       value_type                                                  char_type;
     typedef basic_string<char_type>                               string_type;

     // 28.31.1, construct/copy/destroy:
     explicit match_results(const Allocator& a = Allocator());
     match_results(const match_results& m);
     match_results(match_results&& m) noexcept;
     match_results& operator=(const match_results& m);
     match_results& operator=(match_results&& m);
     ~match_results();

     // 28.32.1, state:
     bool ready() const;

     // 28.32.2, size:
     size_type size() const;
     size_type max_size() const;
     bool empty() const;
```

```
    // 28.32.3, element access:
    difference_type length(size_type sub = 0) const;
    difference_type position(size_type sub = 0) const;
    string_type str(size_type sub = 0) const;
    const_reference operator[](size_type n) const;

    const_reference prefix() const;
    const_reference suffix() const;
    const_iterator begin() const;
    const_iterator end() const;
    const_iterator cbegin() const;
    const_iterator cend() const;

    // 28.32.4, format:
    template <class OutputIter>
     OutputIter
     format(OutputIter out,
            const char_type* fmt_first, const char_type* fmt_last,
            regex_constants::match_flag_type flags =
             regex_constants::format_default) const;
    template <class OutputIter, class ST, class SA>
      OutputIter
      format(OutputIter out,
             const basic_string<char_type, ST, SA>& fmt,
             regex_constants::match_flag_type flags =
               regex_constants::format_default) const;
    template <class ST, class SA>
     basic_string<char_type, ST, SA>
     format(const basic_string<char_type, ST, SA>& fmt,
            regex_constants::match_flag_type flags =
              regex_constants::format_default) const;
    string_type
    format(const char_type* fmt,
           regex_constants::match_flag_type flags =
             regex_constants::format_default) const;

    // 28.32.5, allocator:
    allocator_type get_allocator() const;

    // 28.32.6, swap:
    void swap(match_results& that);
  };
}
```

28.31.1 `match_results` constructors [re.results.const]

1

In all `match_results` constructors, a copy of the `Allocator` argument shall be used for any memory allocation performed by the constructor or member functions during the lifetime of the object.

`match_results(const Allocator& a = Allocator());`

1 *Effects:* Constructs an object of class `match_results`.

2 *Postconditions:* `ready()` returns `false`. `size()` returns `0`.

`match_results(const match_results& m);`

3 *Effects:* Constructs an object of class `match_results`, as a copy of `m`.

`match_results(match_results&& m) noexcept;`

4 *Effects:* Move-constructs an object of class `match_results` from `m` satisfying the same postconditions as Table 142. Additionally, the stored `Allocator` value is move constructed from `m.get_allocator()`.

5 *Throws:* Nothing.

`match_results& operator=(const match_results& m);`

6 *Effects:* Assigns `m` to `*this`. The postconditions of this function are indicated in Table 142.

`match_results& operator=(match_results&& m);`

7 *Effects:* Move-assigns `m` to `*this`. The postconditions of this function are indicated in Table 142.

Table 142 — `match_results` assignment operator effects

Element	Value
`ready()`	`m.ready()`
`size()`	`m.size()`
`str(n)`	`m.str(n)` for all integers `n < m.size()`
`prefix()`	`m.prefix()`
`suffix()`	`m.suffix()`
`(*this)[n]`	`m[n]` for all integers `n < m.size()`
`length(n)`	`m.length(n)` for all integers `n < m.size()`
`position(n)`	`m.position(n)` for all integers `n < m.size()`

28.32.1 `match_results` state [re.results.state]

`bool ready() const;`

1 *Returns:* `true` if `*this` has a fully established result state, otherwise `false`.

28.32.2 `match_results` size [re.results.size]

`size_type size() const;`

1 *Returns:* One plus the number of marked sub-expressions in the regular expression that was matched if `*this` represents the result of a successful match. Otherwise returns `0`. [*Note:* The state of a `match_results` object can be modified only by passing that object to `regex_match` or `regex_search`. Sections 28.35.1 and 28.37.1 specify the effects of those algorithms on their `match_results` arguments. — *end note*]

`size_type max_size() const;`

2 *Returns:* The maximum number of `sub_match` elements that can be stored in `*this`.

`bool empty() const;`

3 *Returns:* `size() == 0`.

28.32.3 `match_results` element access [re.results.acc]

`difference_type length(size_type sub = 0) const;`

1 *Requires:* `ready() == true`.

2 *Returns:* `(*this)[sub].length()`.

`difference_type position(size_type sub = 0) const;`

3 *Requires:* `ready() == true`.

4 *Returns:* The distance from the start of the target sequence to `(*this)[sub].first`.

`string_type str(size_type sub = 0) const;`

5 *Requires:* `ready() == true`.

6 *Returns:* `string_type((*this)[sub])`.

`const_reference operator[](size_type n) const;`

7 *Requires:* `ready() == true`.

8 *Returns:* A reference to the `sub_match` object representing the character sequence that matched marked sub-expression `n`. If `n == 0` then returns a reference to a `sub_match` object representing the character sequence that matched the whole regular expression. If `n >= size()` then returns a `sub_match` object representing an unmatched sub-expression.

`const_reference prefix() const;`

9 *Requires:* `ready() == true`.

10 *Returns:* A reference to the `sub_match` object representing the character sequence from the start of the string being matched/searched to the start of the match found.

`const_reference suffix() const;`

11 *Requires:* `ready() == true`.

12 *Returns:* A reference to the `sub_match` object representing the character sequence from the end of the match found to the end of the string being matched/searched.

```
const_iterator begin() const;
const_iterator cbegin() const;
```

13 *Returns:* A starting iterator that enumerates over all the sub-expressions stored in `*this`.

```
const_iterator end() const;
const_iterator cend() const;
```

14 *Returns:* A terminating iterator that enumerates over all the sub-expressions stored in `*this`.

28.32.4 `match_results` formatting [re.results.form]

```
template <class OutputIter>
  OutputIter format(OutputIter out,
                    const char_type* fmt_first, const char_type* fmt_last,
                    regex_constants::match_flag_type flags =
                      regex_constants::format_default) const;
```

1 *Requires:* ready() == true and `OutputIter` shall satisfy the requirements for an Output Iterator (24.116.1).

2 *Effects:* Copies the character sequence `[fmt_first,fmt_last)` to OutputIter `out`. Replaces each format specifier or escape sequence in the copied range with either the character(s) it represents or the sequence of characters within `*this` to which it refers. The bitmasks specified in `flags` determine which format specifiers and escape sequences are recognized.

3 *Returns:* `out`.

```
template <class OutputIter, class ST, class SA>
  OutputIter format(OutputIter out,
                    const basic_string<char_type, ST, SA>& fmt,
                    regex_constants::match_flag_type flags =
                      regex_constants::format_default) const;
```

4 *Effects:* Equivalent to `return format(out, fmt.data(), fmt.data() + fmt.size(), flags)`.

```
template <class ST, class SA>
  basic_string<char_type, ST, SA>
  format(const basic_string<char_type, ST, SA>& fmt,
         regex_constants::match_flag_type flags =
           regex_constants::format_default) const;
```

5 *Requires:* `ready() == true`.

6 *Effects:* Constructs an empty string `result` of type `basic_string<char_type, ST, SA>` and calls `format(back_inserter(result), fmt, flags)`.

7 *Returns:* `result`.

```
string_type
  format(const char_type* fmt,
         regex_constants::match_flag_type flags =
           regex_constants::format_default) const;
```

8 *Requires:* `ready() == true`.

9 *Effects:* Constructs an empty string `result` of type `string_type` and calls `format(back_inserter(result), fmt, fmt + char_traits<char_type>::length(fmt), flags)`.

10 *Returns:* `result`.

28.32.5 `match_results` allocator [re.results.all]

```
allocator_type get_allocator() const;
```

1 *Returns:* A copy of the Allocator that was passed to the object's constructor or, if that allocator has been replaced, a copy of the most recent replacement.

28.32.6 `match_results` swap [re.results.swap]

```
void swap(match_results& that);
```

1 *Effects:* Swaps the contents of the two sequences.

2 *Postcondition:* `*this` contains the sequence of matched sub-expressions that were in `that`, `that` contains the sequence of matched sub-expressions that were in `*this`.

3 *Complexity:* Constant time.

```
template <class BidirectionalIterator, class Allocator>
  void swap(match_results<BidirectionalIterator, Allocator>& m1,
            match_results<BidirectionalIterator, Allocator>& m2);
```

4 *Effects:* `m1.swap(m2)`.

28.32.7 `match_results` non-member functions [re.results.nonmember]

```
template <class BidirectionalIterator, class Allocator>
bool operator==(const match_results<BidirectionalIterator, Allocator>& m1,
                const match_results<BidirectionalIterator, Allocator>& m2);
```

1 *Returns:* `true` if neither match result is ready, `false` if one match result is ready and the other is not. If both match results are ready, returns `true` only if:

(1.1) — `m1.empty() && m2.empty()`, or

(1.2) — `!m1.empty() && !m2.empty()`, and the following conditions are satisfied:

(1.2.1) — `m1.prefix() == m2.prefix()`,

(1.2.2) — `m1.size() == m2.size() && equal(m1.begin(), m1.end(), m2.begin())`, and

(1.2.3) — `m1.suffix() == m2.suffix()`.

[*Note:* The algorithm `equal` is defined in Clause 25. — *end note*]

```
template <class BidirectionalIterator, class Allocator>
bool operator!=(const match_results<BidirectionalIterator, Allocator>& m1,
                const match_results<BidirectionalIterator, Allocator>& m2);
```

2 *Returns:* `!(m1 == m2)`.

28.33 Regular expression algorithms [re.alg]

28.33.1 exceptions [re.except]

1

The algorithms described in this subclause may throw an exception of type `regex_error`. If such an exception `e` is thrown, `e.code()` shall return either `regex_constants::error_complexity` or `regex_constants::error_stack`.

28.35.1 `regex_match` [re.alg.match]

```
template <class BidirectionalIterator, class Allocator, class charT, class traits>
  bool regex_match(BidirectionalIterator first, BidirectionalIterator last,
                   match_results<BidirectionalIterator, Allocator>& m,
                   const basic_regex<charT, traits>& e,
                   regex_constants::match_flag_type flags =
                     regex_constants::match_default);
```

1 *Requires:* The type `BidirectionalIterator` shall satisfy the requirements of a Bidirectional Iterator (24.165.1).

2

Effects: Determines whether there is a match between the regular expression `e`, and all of the character sequence `[first,last)`. The parameter `flags` is used to control how the expression is matched against the character sequence. Returns `true` if such a match exists, `false` otherwise.

1

Postconditions: `m.ready() == true` in all cases. If the function returns `false`, then the effect on parameter `m` is unspecified except that `m.size()` returns `0` and `m.empty()` returns `true`. Otherwise the effects on parameter `m` are given in Table 143.

Table 143 — Effects of `regex_match` algorithm

Element	Value
`m.size()`	`1 + e.mark_count()`
`m.empty()`	false
`m.prefix().first`	first
`m.prefix().second`	first
`m.prefix().matched`	false
`m.suffix().first`	last
`m.suffix().second`	last
`m.suffix().matched`	false
`m[0].first`	first
`m[0].second`	last
`m[0].matched`	`true`
`m[n].first`	For all integers `0 < n < m.size()`, the start of the sequence that matched sub-expression `n`. Alternatively, if sub-expression `n` did not participate in the match, then `last`.
`m[n].second`	For all integers `0 < n < m.size()`, the end of the sequence that matched sub-expression `n`. Alternatively, if sub-expression `n` did not participate in the match, then `last`.
`m[n].matched`	For all integers `0 < n < m.size()`, `true` if sub-expression `n` participated in the match, `false` otherwise.

```
template <class BidirectionalIterator, class charT, class traits>
  bool regex_match(BidirectionalIterator first, BidirectionalIterator last,
                   const basic_regex<charT, traits>& e,
                   regex_constants::match_flag_type flags =
                     regex_constants::match_default);
```

1 *Effects:* Behaves "as if" by constructing an instance of `match_results<BidirectionalIterator> what`, and then returning the result of `regex_match(first, last, what, e, flags)`.

```
template <class charT, class Allocator, class traits>
  bool regex_match(const charT* str,
                   match_results<const charT*, Allocator>& m,
                   const basic_regex<charT, traits>& e,
                   regex_constants::match_flag_type flags =
                     regex_constants::match_default);
```

2 *Returns:* `regex_match(str, str + char_traits<charT>::length(str), m, e, flags)`.

```
template <class ST, class SA, class Allocator, class charT, class traits>
  bool regex_match(const basic_string<charT, ST, SA>& s,
                   match_results<
                     typename basic_string<charT, ST, SA>::const_iterator,
                     Allocator>& m,
                   const basic_regex<charT, traits>& e,
                   regex_constants::match_flag_type flags =
                     regex_constants::match_default);
```

3 *Returns:* `regex_match(s.begin(), s.end(), m, e, flags)`.

```
template <class charT, class traits>
  bool regex_match(const charT* str,
                   const basic_regex<charT, traits>& e,
                   regex_constants::match_flag_type flags =
                     regex_constants::match_default);
```

4 *Returns:* `regex_match(str, str + char_traits<charT>::length(str), e, flags)`

```
template <class ST, class SA, class charT, class traits>
  bool regex_match(const basic_string<charT, ST, SA>& s,
                   const basic_regex<charT, traits>& e,
                   regex_constants::match_flag_type flags =
                     regex_constants::match_default);
```

5 *Returns:* `regex_match(s.begin(), s.end(), e, flags)`.

28.37.1 `regex_search` [re.alg.search]

```
template <class BidirectionalIterator, class Allocator, class charT, class traits>
  bool regex_search(BidirectionalIterator first, BidirectionalIterator last,
                    match_results<BidirectionalIterator, Allocator>& m,
                    const basic_regex<charT, traits>& e,
                    regex_constants::match_flag_type flags =
                      regex_constants::match_default);
```

1

Requires: Type `BidirectionalIterator` shall satisfy the requirements of a Bidirectional Iterator (24.165.1).

1

Effects: Determines whether there is some sub-sequence within `[first,last)` that matches the regular expression `e`. The parameter `flags` is used to control how the expression is matched against the character sequence. Returns `true` if such a sequence exists, `false` otherwise.

1 *Postconditions:* `m.ready() == true` in all cases. If the function returns `false`, then the effect on parameter `m` is unspecified except that `m.size()` returns `0` and `m.empty()` returns `true`. Otherwise the effects on parameter `m` are given in Table 144.

Table 144 — Effects of `regex_search` algorithm

Element	Value
`m.size()`	`1 + e.mark_count()`
`m.empty()`	`false`
`m.prefix().first`	`first`
`m.prefix().second`	`m[0].first`
`m.prefix().matched`	`m.prefix().first != m.prefix().second`
`m.suffix().first`	`m[0].second`
`m.suffix().second`	`last`
`m.suffix().matched`	`m.suffix().first != m.suffix().second`
`m[0].first`	The start of the sequence of characters that matched the regular expression
`m[0].second`	The end of the sequence of characters that matched the regular expression
`m[0].matched`	`true`

Table 144 — Effects of `regex_search` algorithm (continued)

Element	Value
`m[n].first`	For all integers `0 < n < m.size()`, the start of the sequence that matched sub-expression `n`. Alternatively, if sub-expression `n` did not participate in the match, then `last`.
`m[n].second`	For all integers `0 < n < m.size()`, the end of the sequence that matched sub-expression `n`. Alternatively, if sub-expression `n` did not participate in the match, then `last` .
`m[n].matched`	For all integers `0 < n < m.size()`, `true` if sub-expression `n` participated in the match, `false` otherwise.

```
template <class charT, class Allocator, class traits>
bool regex_search(const charT* str, match_results<const charT*, Allocator>& m,
                  const basic_regex<charT, traits>& e,
                  regex_constants::match_flag_type flags =
                    regex_constants::match_default);
```

2 *Returns:* The result of `regex_search(str, str + char_traits<charT>::length(str), m, e, flags)`.

```
template <class ST, class SA, class Allocator, class charT, class traits>
  bool regex_search(const basic_string<charT, ST, SA>& s,
                    match_results<
                      typename basic_string<charT, ST, SA>::const_iterator,
                      Allocator>& m,
                    const basic_regex<charT, traits>& e,
                    regex_constants::match_flag_type flags =
                      regex_constants::match_default);
```

3 *Returns:* The result of `regex_search(s.begin(), s.end(), m, e, flags)`.

```
template <class BidirectionalIterator, class charT, class traits>
  bool regex_search(BidirectionalIterator first, BidirectionalIterator last,
                    const basic_regex<charT, traits>& e,
                    regex_constants::match_flag_type flags =
                      regex_constants::match_default);
```

1 *Effects:* Behaves "as if" by constructing an object `what` of type `match_results<BidirectionalIterator>` and then returning the result of `regex_search(first, last, what, e, flags)`.

```
template <class charT, class traits>
  bool regex_search(const charT* str,
                    const basic_regex<charT, traits>& e,
                    regex_constants::match_flag_type flags =
                      regex_constants::match_default);
```

2 *Returns:* `regex_search(str, str + char_traits<charT>::length(str), e, flags)`

```
template <class ST, class SA, class charT, class traits>
  bool regex_search(const basic_string<charT, ST, SA>& s,
                    const basic_regex<charT, traits>& e,
                    regex_constants::match_flag_type flags =
                      regex_constants::match_default);
```

3 *Returns:* `regex_search(s.begin(), s.end(), e, flags)`.

28.41.1 regex_replace [re.alg.replace]

```
template <class OutputIterator, class BidirectionalIterator,
    class traits, class charT, class ST, class SA>
  OutputIterator
  regex_replace(OutputIterator out,
                BidirectionalIterator first, BidirectionalIterator last,
                const basic_regex<charT, traits>& e,
                const basic_string<charT, ST, SA>& fmt,
                regex_constants::match_flag_type flags =
                  regex_constants::match_default);
template <class OutputIterator, class BidirectionalIterator,
    class traits, class charT>
  OutputIterator
  regex_replace(OutputIterator out,
                BidirectionalIterator first, BidirectionalIterator last,
                const basic_regex<charT, traits>& e,
                const charT* fmt,
                regex_constants::match_flag_type flags =
                  regex_constants::match_default);
```

1 *Effects:* Constructs a `regex_iterator` object `i` as if by

```
regex_iterator<BidirectionalIterator, charT, traits> i(first, last, e, flags)
```

and uses `i` to enumerate through all of the matches `m` of type `match_results<BidirectionalIterator>` that occur within the sequence `[first,last)`. If no such matches are found and `!(flags & regex_constants::format_no_copy)` then calls

```
out = std::copy(first, last, out)
```

If any matches are found then, for each such match:

(1.1) — If `!(flags & regex_constants::format_no_copy)`, calls

```
out = std::copy(m.prefix().first, m.prefix().second, out)
```

(1.2) — Then calls

```
out = m.format(out, fmt, flags)
```

for the first form of the function and

```
out = m.format(out, fmt, fmt + char_traits<charT>::length(fmt), flags)
```

for the second.

Finally, if such a match is found and `!(flags & regex_constants::format_no_copy)`, calls

```
out = std::copy(last_m.suffix().first, last_m.suffix().second, out)
```

where `last_m` is a copy of the last match found. If `flags & regex_constants::format_first_only` is non-zero then only the first match found is replaced.

2 *Returns:* `out`.

```
template <class traits, class charT, class ST, class SA, class FST, class FSA>
  basic_string<charT, ST, SA>
  regex_replace(const basic_string<charT, ST, SA>& s,
                const basic_regex<charT, traits>& e,
```

```
                const basic_string<charT, FST, FSA>& fmt,
                regex_constants::match_flag_type flags =
                  regex_constants::match_default);
template <class traits, class charT, class ST, class SA>
  basic_string<charT, ST, SA>
  regex_replace(const basic_string<charT, ST, SA>& s,
                const basic_regex<charT, traits>& e,
                const charT* fmt,
                regex_constants::match_flag_type flags =
                  regex_constants::match_default);
```

3 *Effects:* Constructs an empty string `result` of type `basic_string<charT, ST, SA>` and calls `regex_replace(back_inserter(result), s.begin(), s.end(), e, fmt, flags)`.

4 *Returns:* `result`.

```
template <class traits, class charT, class ST, class SA>
  basic_string<charT>
  regex_replace(const charT* s,
                const basic_regex<charT, traits>& e,
                const basic_string<charT, ST, SA>& fmt,
                regex_constants::match_flag_type flags =
                  regex_constants::match_default);
template <class traits, class charT>
  basic_string<charT>
  regex_replace(const charT* s,
                const basic_regex<charT, traits>& e,
                const charT* fmt,
                regex_constants::match_flag_type flags =
                  regex_constants::match_default);
```

5 *Effects:* Constructs an empty string `result` of type `basic_string<charT>` and calls `regex_replace(back_inserter(result), s, s + char_traits<charT>::length(s), e, fmt, flags)`.

6 *Returns:* `result`.

28.42 Regular expression iterators [re.iter]

28.42.1 Class template `regex_iterator` [re.regiter]

1

The class template `regex_iterator` is an iterator adaptor. It represents a new view of an existing iterator sequence, by enumerating all the occurrences of a regular expression within that sequence. A `regex_iterator` uses `regex_search` to find successive regular expression matches within the sequence from which it was constructed. After the iterator is constructed, and every time `operator++` is used, the iterator finds and stores a value of `match_results<BidirectionalIterator>`. If the end of the sequence is reached (`regex_search` returns `false`), the iterator becomes equal to the end-of-sequence iterator value. The default constructor constructs an end-of-sequence iterator object, which is the only legitimate iterator to be used for the end condition. The result of `operator*` on an end-of-sequence iterator is not defined. For any other iterator value a const `match_results<BidirectionalIterator>&` is returned. The result of `operator->` on an end-of-sequence iterator is not defined. For any other iterator value a `const match_results<BidirectionalIterator>*` is returned. It is impossible to store things into `regex_iterator`s. Two end-of-sequence iterators are always equal. An end-of-sequence iterator is not equal to a non-end-of-sequence iterator. Two non-end-of-sequence iterators are equal when they are constructed from the same arguments.

```
namespace std {
```

```
template <class BidirectionalIterator,
          class charT = typename iterator_traits<
            BidirectionalIterator>::value_type,
            class traits = regex_traits<charT> >
class regex_iterator {
public:
   typedef basic_regex<charT, traits>           regex_type;
   typedef match_results<BidirectionalIterator> value_type;
   typedef std::ptrdiff_t                       difference_type;
   typedef const value_type*                    pointer;
   typedef const value_type&                    reference;
   typedef std::forward_iterator_tag            iterator_category;

   regex_iterator();
   regex_iterator(BidirectionalIterator a, BidirectionalIterator b,
                  const regex_type& re,
                  regex_constants::match_flag_type m =
                    regex_constants::match_default);
   regex_iterator(BidirectionalIterator a, BidirectionalIterator b,
                  const regex_type&& re,
                  regex_constants::match_flag_type m =
                    regex_constants::match_default) = delete;
   regex_iterator(const regex_iterator&);
   regex_iterator& operator=(const regex_iterator&);
   bool operator==(const regex_iterator&) const;
   bool operator!=(const regex_iterator&) const;
   const value_type& operator*() const;
   const value_type* operator->() const;
   regex_iterator& operator++();
   regex_iterator operator++(int);
private:
   BidirectionalIterator                begin;  // exposition only
   BidirectionalIterator                end;    // exposition only
   const regex_type*                    pregex; // exposition only
   regex_constants::match_flag_type     flags;  // exposition only
   match_results<BidirectionalIterator> match;  // exposition only
};
}
```

1 An object of type `regex_iterator` that is not an end-of-sequence iterator holds a *zero-length match* if `match[0].matched == true` and `match[0].first == match[0].second`. [*Note:* For example, this can occur when the part of the regular expression that matched consists only of an assertion (such as `'^'`, `'$'`, `'\b'`, `'\B'`). — *end note*]

28.43.0.1 `regex_iterator` constructors [re.regiter.cnstr]

```
regex_iterator();
```

1 *Effects:* Constructs an end-of-sequence iterator.

```
regex_iterator(BidirectionalIterator a, BidirectionalIterator b,
               const regex_type& re,
               regex_constants::match_flag_type m = regex_constants::match_default);
```

2 *Effects:* Initializes `begin` and `end` to `a` and `b`, respectively, sets `pregex` to `&re`, sets `flags` to `m`, then calls `regex_search(begin, end, match, *pregex, flags)`. If this call returns `false` the constructor sets `*this` to the end-of-sequence iterator.

28.43.0.2 `regex_iterator` comparisons [re.regiter.comp]

`bool operator==(const regex_iterator& right) const;`

1 *Returns:* `true` if `*this` and `right` are both end-of-sequence iterators or if the following conditions all hold:

(1.1) — `begin == right.begin`,

(1.2) — `end == right.end`,

(1.3) — `pregex == right.pregex`,

(1.4) — `flags == right.flags`, and

(1.5) — `match[0] == right.match[0]`;

otherwise `false`.

`bool operator!=(const regex_iterator& right) const;`

2 *Returns:* `!(*this == right)`.

28.43.0.3 `regex_iterator` indirection [re.regiter.deref]

`const value_type& operator*() const;`

1 *Returns:* `match`.

`const value_type* operator->() const;`

2 *Returns:* `&match`.

28.43.0.4 `regex_iterator` increment [re.regiter.incr]

`regex_iterator& operator++();`

1 *Effects:* Constructs a local variable `start` of type `BidirectionalIterator` and initializes it with the value of `match[0].second`.

2 If the iterator holds a zero-length match and `start == end` the operator sets `*this` to the end-of-sequence iterator and returns `*this`.

3 Otherwise, if the iterator holds a zero-length match the operator calls `regex_search(start, end, match, *pregex, flags | regex_constants::match_not_null | regex_constants::match_continuous)`. If the call returns `true` the operator returns `*this`. Otherwise the operator increments `start` and continues as if the most recent match was not a zero-length match.

4 If the most recent match was not a zero-length match, the operator sets `flags` to `flags | regex_constants ::match_prev_avail` and calls `regex_search(start, end, match, *pregex, flags)`. If the call returns `false` the iterator sets `*this` to the end-of-sequence iterator. The iterator then returns `*this`.

5 In all cases in which the call to `regex_search` returns `true`, `match.prefix().first` shall be equal to the previous value of `match[0].second`, and for each index `i` in the half-open range `[0, match.size())` for which `match[i].matched` is true, `match[i].position()` shall return `distance(begin, match[i].first)`.

6 [*Note:* This means that `match[i].position()` gives the offset from the beginning of the target sequence, which is often not the same as the offset from the sequence passed in the call to `regex_search`. — *end note*]

7 It is unspecified how the implementation makes these adjustments.

8 [*Note:* This means that a compiler may call an implementation-specific search function, in which case a user-defined specialization of `regex_search` will not be called. — *end note*]

```
regex_iterator operator++(int);
```

9 *Effects:*

```
regex_iterator tmp = *this;
++(*this);
return tmp;
```

28.43.1 Class template `regex_token_iterator` [re.tokiter]

1 The class template `regex_token_iterator` is an iterator adaptor; that is to say it represents a new view of an existing iterator sequence, by enumerating all the occurrences of a regular expression within that sequence, and presenting one or more sub-expressions for each match found. Each position enumerated by the iterator is a `sub_match` class template instance that represents what matched a particular sub-expression within the regular expression.

2 When class `regex_token_iterator` is used to enumerate a single sub-expression with index -1 the iterator performs field splitting: that is to say it enumerates one sub-expression for each section of the character container sequence that does not match the regular expression specified.

3 After it is constructed, the iterator finds and stores a value `regex_iterator<BidirectionalIterator> position` and sets the internal count `N` to zero. It also maintains a sequence `subs` which contains a list of the sub-expressions which will be enumerated. Every time `operator++` is used the count `N` is incremented; if `N` exceeds or equals `subs.size()`, then the iterator increments member `position` and sets count `N` to zero.

4

If the end of sequence is reached (`position` is equal to the end of sequence iterator), the iterator becomes equal to the end-of-sequence iterator value, unless the sub-expression being enumerated has index -1, in which case the iterator enumerates one last sub-expression that contains all the characters from the end of the last regular expression match to the end of the input sequence being enumerated, provided that this would not be an empty sub-expression.

1 The default constructor constructs an end-of-sequence iterator object, which is the only legitimate iterator to be used for the end condition. The result of `operator*` on an end-of-sequence iterator is not defined. For any other iterator value a `const sub_match<BidirectionalIterator>&` is returned. The result of `operator->` on an end-of-sequence iterator is not defined. For any other iterator value a `const sub_match<BidirectionalIterator>*` is returned.

2 It is impossible to store things into `regex_token_iterators`. Two end-of-sequence iterators are always equal. An end-of-sequence iterator is not equal to a non-end-of-sequence iterator. Two non-end-of-sequence iterators are equal when they are constructed from the same arguments.

```
namespace std {
  template <class BidirectionalIterator,
            class charT = typename iterator_traits<
              BidirectionalIterator>::value_type,
              class traits = regex_traits<charT> >
  class regex_token_iterator  {
  public:
    typedef basic_regex<charT, traits>       regex_type;
    typedef sub_match<BidirectionalIterator> value_type;
    typedef std::ptrdiff_t                   difference_type;
    typedef const value_type*                pointer;
    typedef const value_type&                reference;
    typedef std::forward_iterator_tag        iterator_category;

    regex_token_iterator();
```

```
  regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
                       const regex_type& re,
                       int submatch = 0,
                       regex_constants::match_flag_type m =
                         regex_constants::match_default);
  regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
                       const regex_type& re,
                       const std::vector<int>& submatches,
                       regex_constants::match_flag_type m =
                         regex_constants::match_default);
  regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
                       const regex_type& re,
                       initializer_list<int> submatches,
                       regex_constants::match_flag_type m =
                         regex_constants::match_default);
  template <std::size_t N>
    regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
                       const regex_type& re,
                       const int (&submatches)[N],
                       regex_constants::match_flag_type m =
                         regex_constants::match_default);
  regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
                        const regex_type&& re,
                        int submatch = 0,
                        regex_constants::match_flag_type m =
                          regex_constants::match_default) = delete;
  regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
                        const regex_type&& re,
                        const std::vector<int>& submatches,
                        regex_constants::match_flag_type m =
                          regex_constants::match_default) = delete;
  regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
                        const regex_type&& re,
                        initializer_list<int> submatches,
                        regex_constants::match_flag_type m =
                          regex_constants::match_default) = delete;
  template <std::size_t N>
  regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
                        const regex_type&& re,
                        const int (&submatches)[N],
                        regex_constants::match_flag_type m =
                          regex_constants::match_default) = delete;
  regex_token_iterator(const regex_token_iterator&);
  regex_token_iterator& operator=(const regex_token_iterator&);
  bool operator==(const regex_token_iterator&) const;
  bool operator!=(const regex_token_iterator&) const;
  const value_type& operator*() const;
  const value_type* operator->() const;
  regex_token_iterator& operator++();
  regex_token_iterator operator++(int);
private:
  typedef
    regex_iterator<BidirectionalIterator, charT, traits> position_iterator; // exposition only
  position_iterator position;                                               // exposition only
  const value_type* result;                                                 // exposition only
```

```
    value_type suffix;                                    // exposition only
    std::size_t N;                                        // exposition only
    std::vector<int> subs;                                // exposition only
  };
}
```

3 A *suffix iterator* is a `regex_token_iterator` object that points to a final sequence of characters at the end of the target sequence. In a suffix iterator the member `result` holds a pointer to the data member `suffix`, the value of the member `suffix.match` is `true`, `suffix.first` points to the beginning of the final sequence, and `suffix.second` points to the end of the final sequence.

4 [*Note:* For a suffix iterator, data member `suffix.first` is the same as the end of the last match found, and `suffix.second` is the same as the end of the target sequence — *end note*]

5 The *current match* is `(*position).prefix()` if `subs[N] == -1`, or `(*position)[subs[N]]` for any other value of `subs[N]`.

28.44.0.1 `regex_token_iterator` constructors [re.tokiter.cnstr]

```
regex_token_iterator();
```

1 *Effects:* Constructs the end-of-sequence iterator.

```
regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
                     const regex_type& re,
                     int submatch = 0,
                     regex_constants::match_flag_type m =
                      regex_constants::match_default);

regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
                     const regex_type& re,
                     const std::vector<int>& submatches,
                     regex_constants::match_flag_type m =
                      regex_constants::match_default);

regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
                     const regex_type& re,
                     initializer_list<int> submatches,
                     regex_constants::match_flag_type m =
                       regex_constants::match_default);

template <std::size_t N>
  regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
                     const regex_type& re,
                     const int (&submatches)[N],
                     regex_constants::match_flag_type m =
                      regex_constants::match_default);
```

2 *Requires:* Each of the initialization values of `submatches` shall be `>= -1`.

3 *Effects:* The first constructor initializes the member `subs` to hold the single value `submatch`. The second constructor initializes the member `subs` to hold a copy of the argument `submatches`. The third and fourth constructors initialize the member `subs` to hold a copy of the sequence of integer values pointed to by the iterator range `[submatches.begin(),submatches.end())` and `[&submatches,&submatches + N)`, respectively.

4 Each constructor then sets `N` to 0, and `position` to `position_iterator(a, b, re, m)`. If `position` is not an end-of-sequence iterator the constructor sets `result` to the address of the current match.

Otherwise if any of the values stored in `subs` is equal to -1 the constructor sets `*this` to a suffix iterator that points to the range `[a,b)`, otherwise the constructor sets `*this` to an end-of-sequence iterator.

28.44.0.2 `regex_token_iterator` comparisons [re.tokiter.comp]

```
bool operator==(const regex_token_iterator& right) const;
```

1 *Returns:* `true` if `*this` and `right` are both end-of-sequence iterators, or if `*this` and `right` are both suffix iterators and `suffix == right.suffix`; otherwise returns `false` if `*this` or `right` is an end-of-sequence iterator or a suffix iterator. Otherwise returns `true` if `position == right.position`, `N == right.N`, and `subs == right.subs`. Otherwise returns `false`.

```
bool operator!=(const regex_token_iterator& right) const;
```

2 *Returns:* `!(*this == right)`.

28.44.0.3 `regex_token_iterator` indirection [re.tokiter.deref]

```
const value_type& operator*() const;
```

1 *Returns:* `*result`.

```
const value_type* operator->() const;
```

2 *Returns:* `result`.

28.44.0.4 `regex_token_iterator` increment [re.tokiter.incr]

```
regex_token_iterator& operator++();
```

1 *Effects:* Constructs a local variable `prev` of type `position_iterator`, initialized with the value of `position`.

2 If `*this` is a suffix iterator, sets `*this` to an end-of-sequence iterator.

3 Otherwise, if `N + 1 < subs.size()`, increments `N` and sets `result` to the address of the current match.

4 Otherwise, sets `N` to 0 and increments `position`. If `position` is not an end-of-sequence iterator the operator sets `result` to the address of the current match.

5 Otherwise, if any of the values stored in `subs` is equal to -1 and `prev->suffix().length()` is not 0 the operator sets `*this` to a suffix iterator that points to the range `[prev->suffix().first, prev->suffix().second)`.

6 Otherwise, sets `*this` to an end-of-sequence iterator.

Returns: `*this`

```
regex_token_iterator& operator++(int);
```

7 *Effects:* Constructs a copy `tmp` of `*this`, then calls `++(*this)`.

8 *Returns:* `tmp`.

28.45 Modified ECMAScript regular expression grammar [re.grammar]

1

The regular expression grammar recognized by `basic_regex` objects constructed with the ECMAScript flag is that specified by ECMA-262, except as specified below.

1 Objects of type specialization of `basic_regex` store within themselves a default-constructed instance of their `traits` template parameter, henceforth referred to as `traits_inst`. This `traits_inst` object is used

to support localization of the regular expression; `basic_regex` member functions shall not call any locale dependent C or C++ API, including the formatted string input functions. Instead they shall call the appropriate traits member function to achieve the required effect.

2 The following productions within the ECMAScript grammar are modified as follows:

```
ClassAtom ::
  -
  ClassAtomNoDash
  ClassAtomExClass
  ClassAtomCollatingElement
  ClassAtomEquivalence
```

3 The following new productions are then added:

```
ClassAtomExClass ::
  [: ClassName :]

ClassAtomCollatingElement ::
  [. ClassName .]

ClassAtomEquivalence ::
  [= ClassName =]

ClassName ::
  ClassNameCharacter
  ClassNameCharacter ClassName

ClassNameCharacter ::
  SourceCharacter but not one of "." "=" ":"
```

4 The productions `ClassAtomExClass`, `ClassAtomCollatingElement` and `ClassAtomEquivalence` provide functionality equivalent to that of the same features in regular expressions in POSIX.

5 The regular expression grammar may be modified by any `regex_constants::syntax_option_type` flags specified when constructing an object of type specialization of `basic_regex` according to the rules in Table 138.

6 A `ClassName` production, when used in `ClassAtomExClass`, is not valid if `traits_inst.lookup_classname` returns zero for that name. The names recognized as valid `ClassName`s are determined by the type of the traits class, but at least the following names shall be recognized: `alnum`, `alpha`, `blank`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper`, `xdigit`, `d`, `s`, `w`. In addition the following expressions shall be equivalent:

```
\d and [[:digit:]]

\D and [^[:digit:]]

\s and [[:space:]]

\S and [^[:space:]]

\w and [_[:alnum:]]

\W and [^_[:alnum:]]
```

7 A `ClassName` production when used in a `ClassAtomCollatingElement` production is not valid if the value returned by `traits_inst.lookup_collatename` for that name is an empty string.

8 The results from multiple calls to `traits_inst.lookup_classname` can be bitwise OR'ed together and subsequently passed to `traits_inst.isctype`.

9 A `ClassName` production when used in a `ClassAtomEquivalence` production is not valid if the value returned by `traits_inst.lookup_collatename` for that name is an empty string or if the value returned by `traits_inst.transform_primary` for the result of the call to `traits_inst.lookup_collatename` is an empty string.

10 When the sequence of characters being transformed to a finite state machine contains an invalid class name the translator shall throw an exception object of type `regex_error`.

11 If the *CV* of a *UnicodeEscapeSequence* is greater than the largest value that can be held in an object of type `charT` the translator shall throw an exception object of type `regex_error`. [*Note:* This means that values of the form `"uxxxx"` that do not fit in a character are invalid. — *end note*]

12 Where the regular expression grammar requires the conversion of a sequence of characters to an integral value, this is accomplished by calling `traits_inst.value`.

13

The behavior of the internal finite state machine representation when used to match a sequence of characters is as described in ECMA-262. The behavior is modified according to any match_flag_type flags 28.10.1 specified when using the regular expression object in one of the regular expression algorithms 28.33. The behavior is also localized by interaction with the traits class template parameter as follows:

— During matching of a regular expression finite state machine against a sequence of characters, two characters `c` and `d` are compared using the following rules:

1. if `(flags() & regex_constants::icase)` the two characters are equal if `traits_inst.translate_nocase(c) == traits_inst.translate_nocase(d)`;
2. otherwise, if `flags() & regex_constants::collate` the two characters are equal if `traits_inst.translate(c) == traits_inst.translate(d)`;
3. otherwise, the two characters are equal if `c == d`.

— During matching of a regular expression finite state machine against a sequence of characters, comparison of a collating element range `c1-c2` against a character `c` is conducted as follows: if `flags() & regex_constants ::collate` is false then the character `c` is matched if `c1 <= c && c <= c2`, otherwise `c` is matched in accordance with the following algorithm:

```
string_type str1 = string_type(1,
  flags() & icase ?
    traits_inst.translate_nocase(c1) : traits_inst.translate(c1);
string_type str2 = string_type(1,
  flags() & icase ?
    traits_inst.translate_nocase(c2) : traits_inst.translate(c2);
string_type str = string_type(1,
  flags() & icase ?
    traits_inst.translate_nocase(c) : traits_inst.translate(c);
return traits_inst.transform(str1.begin(), str1.end())
      <= traits_inst.transform(str.begin(), str.end())
  && traits_inst.transform(str.begin(), str.end())
      <= traits_inst.transform(str2.begin(), str2.end());
```

— During matching of a regular expression finite state machine against a sequence of characters, testing whether a collating element is a member of a primary equivalence class is conducted by first converting the collating element and the equivalence class to sort keys using `traits::transform_primary`, and then comparing the sort keys for equality.

— During matching of a regular expression finite state machine against a sequence of characters, a character `c` is a member of a character class designated by an iterator range `[first,last)` if `traits_inst.isctype(c, traits_inst.lookup_classname(first, last, flags() & icase))` is `true`.

29 Atomic operations library [atomics]

29.1 General [atomics.general]

1 This Clause describes components for fine-grained atomic access. This access is provided via operations on atomic objects.

2 The following subclauses describe atomics requirements and components for types and operations, as summarized below.

Table 145 — Atomics library summary

Subclause		Header(s)
29.3	Order and Consistency	
29.31	Lock-free Property	
29.32	Atomic Types	`<atomic>`
29.50	Operations on Atomic Types	
29.77	Flag Type and Operations	
29.85	Fences	

29.2 Header `<atomic>` synopsis [atomics.syn]

```
namespace std {
  // 29.3, order and consistency
  enum memory_order;
  template <class T>
    T kill_dependency(T y) noexcept;

  // 29.31, lock-free property
  #define ATOMIC_BOOL_LOCK_FREE unspecified
  #define ATOMIC_CHAR_LOCK_FREE unspecified
  #define ATOMIC_CHAR16_T_LOCK_FREE unspecified
  #define ATOMIC_CHAR32_T_LOCK_FREE unspecified
  #define ATOMIC_WCHAR_T_LOCK_FREE unspecified
  #define ATOMIC_SHORT_LOCK_FREE unspecified
  #define ATOMIC_INT_LOCK_FREE unspecified
  #define ATOMIC_LONG_LOCK_FREE unspecified
  #define ATOMIC_LLONG_LOCK_FREE unspecified
  #define ATOMIC_POINTER_LOCK_FREE unspecified

  // 29.32, generic types
  template<class T> struct atomic;
  template<> struct atomic<integral>;
  template<class T> struct atomic<T*>;

  // 29.50.1, general operations on atomic types
  // In the following declarations, atomic-type is either
  // atomic<T> or a named base class for T from
  // Table 146 or inferred from Table 147 or from bool.
  // If it is atomic<T>, then the declaration is a template
  // declaration prefixed with template <class T>.
```

```
bool atomic_is_lock_free(const volatile atomic-type*) noexcept;
bool atomic_is_lock_free(const atomic-type*) noexcept;
void atomic_init(volatile atomic-type*, T) noexcept;
void atomic_init(atomic-type*, T) noexcept;
void atomic_store(volatile atomic-type*, T) noexcept;
void atomic_store(atomic-type*, T) noexcept;
void atomic_store_explicit(volatile atomic-type*, T, memory_order) noexcept;
void atomic_store_explicit(atomic-type*, T, memory_order) noexcept;
T atomic_load(const volatile atomic-type*) noexcept;
T atomic_load(const atomic-type*) noexcept;
T atomic_load_explicit(const volatile atomic-type*, memory_order) noexcept;
T atomic_load_explicit(const atomic-type*, memory_order) noexcept;
T atomic_exchange(volatile atomic-type*, T) noexcept;
T atomic_exchange(atomic-type*, T) noexcept;
T atomic_exchange_explicit(volatile atomic-type*, T, memory_order) noexcept;
T atomic_exchange_explicit(atomic-type*, T, memory_order) noexcept;
bool atomic_compare_exchange_weak(volatile atomic-type*, T*, T) noexcept;
bool atomic_compare_exchange_weak(atomic-type*, T*, T) noexcept;
bool atomic_compare_exchange_strong(volatile atomic-type*, T*, T) noexcept;
bool atomic_compare_exchange_strong(atomic-type*, T*, T) noexcept;
bool atomic_compare_exchange_weak_explicit(volatile atomic-type*, T*, T,
  memory_order, memory_order) noexcept;
bool atomic_compare_exchange_weak_explicit(atomic-type*, T*, T,
  memory_order, memory_order) noexcept;
bool atomic_compare_exchange_strong_explicit(volatile atomic-type*, T*, T,
  memory_order, memory_order) noexcept;
bool atomic_compare_exchange_strong_explicit(atomic-type*, T*, T,
  memory_order, memory_order) noexcept;

// 29.50.2, templated operations on atomic types
template <class T>
  T atomic_fetch_add(volatile atomic<T>*, T) noexcept;
template <class T>
  T atomic_fetch_add(atomic<T>*, T) noexcept;
template <class T>
  T atomic_fetch_add_explicit(volatile atomic<T>*, T, memory_order) noexcept;
template <class T>
  T atomic_fetch_add_explicit(atomic<T>*, T, memory_order) noexcept;
template <class T>
  T atomic_fetch_sub(volatile atomic<T>*, T) noexcept;
template <class T>
  T atomic_fetch_sub(atomic<T>*, T) noexcept;
template <class T>
  T atomic_fetch_sub_explicit(volatile atomic<T>*, T, memory_order) noexcept;
template <class T>
  T atomic_fetch_sub_explicit(atomic<T>*, T, memory_order) noexcept;
template <class T>
  T atomic_fetch_and(volatile atomic<T>*, T) noexcept;
template <class T>
  T atomic_fetch_and(atomic<T>*, T) noexcept;
template <class T>
  T atomic_fetch_and_explicit(volatile atomic<T>*, T, memory_order) noexcept;
template <class T>
  T atomic_fetch_and_explicit(atomic<T>*, T, memory_order) noexcept;
template <class T>
```

```
  T atomic_fetch_or(volatile atomic<T>*, T) noexcept;
template <class T>
  T atomic_fetch_or(atomic<T>*, T) noexcept;
template <class T>
  T atomic_fetch_or_explicit(volatile atomic<T>*, T, memory_order) noexcept;
template <class T>
  T atomic_fetch_or_explicit(atomic<T>*, T, memory_order) noexcept;
template <class T>
  T atomic_fetch_xor(volatile atomic<T>*, T) noexcept;
template <class T>
  T atomic_fetch_xor(atomic<T>*, T) noexcept;
template <class T>
  T atomic_fetch_xor_explicit(volatile atomic<T>*, T, memory_order) noexcept;
template <class T>
  T atomic_fetch_xor_explicit(atomic<T>*, T, memory_order) noexcept;

// 29.50.3, arithmetic operations on atomic types
// In the following declarations, atomic-integral is either
// atomic<T> or a named base class for T from
// Table 146 or inferred from Table 147.
// If it is atomic<T>, then the declaration is a template
// specialization declaration prefixed with template <>.

integral atomic_fetch_add(volatile atomic-integral*, integral) noexcept;
integral atomic_fetch_add(atomic-integral*, integral) noexcept;
integral atomic_fetch_add_explicit(volatile atomic-integral*, integral, memory_order) noexcept;
integral atomic_fetch_add_explicit(atomic-integral*, integral, memory_order) noexcept;
integral atomic_fetch_sub(volatile atomic-integral*, integral) noexcept;
integral atomic_fetch_sub(atomic-integral*, integral) noexcept;
integral atomic_fetch_sub_explicit(volatile atomic-integral*, integral, memory_order) noexcept;
integral atomic_fetch_sub_explicit(atomic-integral*, integral, memory_order) noexcept;
integral atomic_fetch_and(volatile atomic-integral*, integral) noexcept;
integral atomic_fetch_and(atomic-integral*, integral) noexcept;
integral atomic_fetch_and_explicit(volatile atomic-integral*, integral, memory_order) noexcept;
integral atomic_fetch_and_explicit(atomic-integral*, integral, memory_order) noexcept;
integral atomic_fetch_or(volatile atomic-integral*, integral) noexcept;
integral atomic_fetch_or(atomic-integral*, integral) noexcept;
integral atomic_fetch_or_explicit(volatile atomic-integral*, integral, memory_order) noexcept;
integral atomic_fetch_or_explicit(atomic-integral*, integral, memory_order) noexcept;
integral atomic_fetch_xor(volatile atomic-integral*, integral) noexcept;
integral atomic_fetch_xor(atomic-integral*, integral) noexcept;
integral atomic_fetch_xor_explicit(volatile atomic-integral*, integral, memory_order) noexcept;
integral atomic_fetch_xor_explicit(atomic-integral*, integral, memory_order) noexcept;

// 29.51.1, partial specializations for pointers

template <class T>
  T* atomic_fetch_add(volatile atomic<T*>*, ptrdiff_t) noexcept;
template <class T>
  T* atomic_fetch_add(atomic<T*>*, ptrdiff_t) noexcept;
template <class T>
  T* atomic_fetch_add_explicit(volatile atomic<T*>*, ptrdiff_t, memory_order) noexcept;
template <class T>
  T* atomic_fetch_add_explicit(atomic<T*>*, ptrdiff_t, memory_order) noexcept;
template <class T>
```

```
    T* atomic_fetch_sub(volatile atomic<T*>*, ptrdiff_t) noexcept;
  template <class T>
    T* atomic_fetch_sub(atomic<T*>*, ptrdiff_t) noexcept;
  template <class T>
    T* atomic_fetch_sub_explicit(volatile atomic<T*>*, ptrdiff_t, memory_order) noexcept;
  template <class T>
    T* atomic_fetch_sub_explicit(atomic<T*>*, ptrdiff_t, memory_order) noexcept;

  // 29.51.2, initialization
  #define ATOMIC_VAR_INIT(value) see below

  // 29.77, flag type and operations
  struct atomic_flag;
  bool atomic_flag_test_and_set(volatile atomic_flag*) noexcept;
  bool atomic_flag_test_and_set(atomic_flag*) noexcept;
  bool atomic_flag_test_and_set_explicit(volatile atomic_flag*, memory_order) noexcept;
  bool atomic_flag_test_and_set_explicit(atomic_flag*, memory_order) noexcept;
  void atomic_flag_clear(volatile atomic_flag*) noexcept;
  void atomic_flag_clear(atomic_flag*) noexcept;
  void atomic_flag_clear_explicit(volatile atomic_flag*, memory_order) noexcept;
  void atomic_flag_clear_explicit(atomic_flag*, memory_order) noexcept;
  #define ATOMIC_FLAG_INIT see below

  // 29.85, fences
  extern "C" void atomic_thread_fence(memory_order) noexcept;
  extern "C" void atomic_signal_fence(memory_order) noexcept;
}
```

29.3 Order and consistency [atomics.order]

```
namespace std {
  typedef enum memory_order {
    memory_order_relaxed, memory_order_consume, memory_order_acquire,
    memory_order_release, memory_order_acq_rel, memory_order_seq_cst
  } memory_order;
}
```

1

The enumeration `memory_order` specifies the detailed regular (non-atomic) memory synchronization order as defined in 1.859 and may provide for operation ordering. Its enumerated values and their meanings are as follows:

— `memory_order_relaxed`: no operation orders memory.

— `memory_order_release`, `memory_order_acq_rel`, and `memory_order_seq_cst`: a store operation performs a release operation on the affected memory location.

— `memory_order_consume`: a load operation performs a consume operation on the affected memory location.

— `memory_order_acquire`, `memory_order_acq_rel`, and `memory_order_seq_cst`: a load operation performs an acquire operation on the affected memory location.

[*Note:* Atomic operations specifying `memory_order_relaxed` are relaxed with respect to memory ordering. Implementations must still guarantee that any given atomic access to a particular atomic object be indivisible with respect to all other atomic accesses to that object. — *end note*]

1

An atomic operation A that performs a release operation on an atomic object M synchronizes with an atomic operation B that performs an acquire operation on M and takes its value from any side effect in the release sequence headed by A.

1

There shall be a single total order S on all `memory_order_seq_cst` operations, consistent with the "happens before" order and modification orders for all affected locations, such that each `memory_order_seq_cst` operation B that loads a value from an atomic object M observes one of the following values:

— the result of the last modification A of M that precedes B in S, if it exists, or

— if A exists, the result of some modification of M that is not `memory_order_seq_cst` and that does not happen before A, or

— if A does not exist, the result of some modification of M that is not `memory_order_seq_cst`.

[*Note:* Although it is not explicitly required that S include locks, it can always be extended to an order that does include lock and unlock operations, since the ordering between those is already included in the "happens before" ordering. — *end note*]

1

For an atomic operation B that reads the value of an atomic object M, if there is a `memory_order_seq_cst` fence X sequenced before B, then B observes either the last `memory_order_seq_cst` modification of M preceding X in the total order S or a later modification of M in its modification order.

1 For atomic operations A and B on an atomic object M, where A modifies M and B takes its value, if there is a `memory_order_seq_cst` fence X such that A is sequenced before X and B follows X in S, then B observes either the effects of A or a later modification of M in its modification order.

2

For atomic operations A and B on an atomic object M, where A modifies M and B takes its value, if there are `memory_order_seq_cst` fences X and Y such that A is sequenced before X, Y is sequenced before B, and X precedes Y in S, then B observes either the effects of A or a later modification of M in its modification order.

1 For atomic modifications A and B of an atomic object M, B occurs later than A in the modification order of M if:

(1.1) — there is a `memory_order_seq_cst` fence X such that A is sequenced before X, and X precedes B in S, or

(1.2) — there is a `memory_order_seq_cst` fence Y such that Y is sequenced before B, and A precedes Y in S, or

(1.3) — there are `memory_order_seq_cst` fences X and Y such that A is sequenced before X, Y is sequenced before B, and X precedes Y in S.

2

[*Note:* `memory_order_seq_cst` ensures sequential consistency only for a program that is free of data races and uses exclusively `memory_order_seq_cst` operations. Any use of weaker ordering will invalidate this guarantee unless extreme care is used. In particular, `memory_order_seq_cst` fences ensure a total order only for the fences themselves. Fences cannot, in general, be used to restore sequential consistency for atomic operations with weaker ordering specifications. — *end note*]

1 Implementations should ensure that no "out-of-thin-air" values are computed that circularly depend on their own computation.

[*Note:* For example, with `x` and `y` initially zero,

```
// Thread 1:
r1 = y.load(memory_order_relaxed);
x.store(r1, memory_order_relaxed);

// Thread 2:
r2 = x.load(memory_order_relaxed);
y.store(r2, memory_order_relaxed);
```

should not produce `r1 == r2 == 42`, since the store of 42 to `y` is only possible if the store to `x` stores `42`, which circularly depends on the store to `y` storing `42`. Note that without this restriction, such an execution is possible. — *end note*]

2 [*Note:* The recommendation similarly disallows `r1 == r2 == 42` in the following example, with `x` and `y` again initially zero:

```
// Thread 1:
r1 = x.load(memory_order_relaxed);
if (r1 == 42) y.store(42, memory_order_relaxed);

// Thread 2:
r2 = y.load(memory_order_relaxed);
if (r2 == 42) x.store(42, memory_order_relaxed);
```

— *end note*]

3

Atomic read-modify-write operations shall always read the last value (in the modification order) written before the write associated with the read-modify-write operation.

1

Implementations should make atomic stores visible to atomic loads within a reasonable amount of time.

```
template <class T>
  T kill_dependency(T y) noexcept;
```

1 *Effects:* The argument does not carry a dependency to the return value (1.859).

2 *Returns:* `y`.

29.31 Lock-free property [atomics.lockfree]

```
#define ATOMIC_BOOL_LOCK_FREE unspecified
#define ATOMIC_CHAR_LOCK_FREE unspecified
#define ATOMIC_CHAR16_T_LOCK_FREE unspecified
#define ATOMIC_CHAR32_T_LOCK_FREE unspecified
#define ATOMIC_WCHAR_T_LOCK_FREE unspecified
#define ATOMIC_SHORT_LOCK_FREE unspecified
#define ATOMIC_INT_LOCK_FREE unspecified
#define ATOMIC_LONG_LOCK_FREE unspecified
#define ATOMIC_LLONG_LOCK_FREE unspecified
#define ATOMIC_POINTER_LOCK_FREE unspecified
```

1 The `ATOMIC_..._LOCK_FREE` macros indicate the lock-free property of the corresponding atomic types, with the signed and unsigned variants grouped together. The properties also apply to the corresponding (partial) specializations of the `atomic` template. A value of 0 indicates that the types are never lock-free. A value of 1 indicates that the types are sometimes lock-free. A value of 2 indicates that the types are always lock-free.

2 The function `atomic_is_lock_free` (29.50) indicates whether the object is lock-free. In any given program execution, the result of the lock-free query shall be consistent for all pointers of the same type.

3 [*Note:* Operations that are lock-free should also be address-free. That is, atomic operations on the same memory location via two different addresses will communicate atomically. The implementation should not depend on any per-process state. This restriction enables communication by memory that is mapped into a process more than once and by memory that is shared between two processes. — *end note*]

29.32 Atomic types [atomics.types.generic]

```
namespace std {
  template <class T> struct atomic {
    bool is_lock_free() const volatile noexcept;
    bool is_lock_free() const noexcept;
    void store(T, memory_order = memory_order_seq_cst) volatile noexcept;
    void store(T, memory_order = memory_order_seq_cst) noexcept;
    T load(memory_order = memory_order_seq_cst) const volatile noexcept;
    T load(memory_order = memory_order_seq_cst) const noexcept;
    operator T() const volatile noexcept;
    operator T() const noexcept;
    T exchange(T, memory_order = memory_order_seq_cst) volatile noexcept;
    T exchange(T, memory_order = memory_order_seq_cst) noexcept;
    bool compare_exchange_weak(T&, T, memory_order, memory_order) volatile noexcept;
    bool compare_exchange_weak(T&, T, memory_order, memory_order) noexcept;
    bool compare_exchange_strong(T&, T, memory_order, memory_order) volatile noexcept;
    bool compare_exchange_strong(T&, T, memory_order, memory_order) noexcept;
    bool compare_exchange_weak(T&, T, memory_order = memory_order_seq_cst) volatile noexcept;
    bool compare_exchange_weak(T&, T, memory_order = memory_order_seq_cst) noexcept;
    bool compare_exchange_strong(T&, T, memory_order = memory_order_seq_cst) volatile noexcept;
    bool compare_exchange_strong(T&, T, memory_order = memory_order_seq_cst) noexcept;

    atomic() noexcept = default;
    constexpr atomic(T) noexcept;
    atomic(const atomic&) = delete;
    atomic& operator=(const atomic&) = delete;
    atomic& operator=(const atomic&) volatile = delete;
    T operator=(T) volatile noexcept;
    T operator=(T) noexcept;
  };

  template <> struct atomic<integral> {
    bool is_lock_free() const volatile noexcept;
    bool is_lock_free() const noexcept;
    void store(integral, memory_order = memory_order_seq_cst) volatile noexcept;
    void store(integral, memory_order = memory_order_seq_cst) noexcept;
    integral load(memory_order = memory_order_seq_cst) const volatile noexcept;
    integral load(memory_order = memory_order_seq_cst) const noexcept;
    operator integral() const volatile noexcept;
    operator integral() const noexcept;
    integral exchange(integral, memory_order = memory_order_seq_cst) volatile noexcept;
    integral exchange(integral, memory_order = memory_order_seq_cst) noexcept;
    bool compare_exchange_weak(integral&, integral, memory_order, memory_order) volatile noexcept;
    bool compare_exchange_weak(integral&, integral, memory_order, memory_order) noexcept;
    bool compare_exchange_strong(integral&, integral, memory_order, memory_order) volatile noexcept;
    bool compare_exchange_strong(integral&, integral, memory_order, memory_order) noexcept;
    bool compare_exchange_weak(integral&, integral, memory_order = memory_order_seq_cst) volatile noexcept;
```

```
  bool compare_exchange_weak(integral&, integral, memory_order = memory_order_seq_cst) noexcept;
  bool compare_exchange_strong(integral&, integral, memory_order = memory_order_seq_cst) volatile noexcept;
  bool compare_exchange_strong(integral&, integral, memory_order = memory_order_seq_cst) noexcept;
  integral fetch_add(integral, memory_order = memory_order_seq_cst) volatile noexcept;
  integral fetch_add(integral, memory_order = memory_order_seq_cst) noexcept;
  integral fetch_sub(integral, memory_order = memory_order_seq_cst) volatile noexcept;
  integral fetch_sub(integral, memory_order = memory_order_seq_cst) noexcept;
  integral fetch_and(integral, memory_order = memory_order_seq_cst) volatile noexcept;
  integral fetch_and(integral, memory_order = memory_order_seq_cst) noexcept;
  integral fetch_or(integral, memory_order = memory_order_seq_cst) volatile noexcept;
  integral fetch_or(integral, memory_order = memory_order_seq_cst) noexcept;
  integral fetch_xor(integral, memory_order = memory_order_seq_cst) volatile noexcept;
  integral fetch_xor(integral, memory_order = memory_order_seq_cst) noexcept;

  atomic() noexcept = default;
  constexpr atomic(integral) noexcept;
  atomic(const atomic&) = delete;
  atomic& operator=(const atomic&) = delete;
  atomic& operator=(const atomic&) volatile = delete;
  integral operator=(integral) volatile noexcept;
  integral operator=(integral) noexcept;

  integral operator++(int) volatile noexcept;
  integral operator++(int) noexcept;
  integral operator--(int) volatile noexcept;
  integral operator--(int) noexcept;
  integral operator++() volatile noexcept;
  integral operator++() noexcept;
  integral operator--() volatile noexcept;
  integral operator--() noexcept;
  integral operator+=(integral) volatile noexcept;
  integral operator+=(integral) noexcept;
  integral operator-=(integral) volatile noexcept;
  integral operator-=(integral) noexcept;
  integral operator&=(integral) volatile noexcept;
  integral operator&=(integral) noexcept;
  integral operator|=(integral) volatile noexcept;
  integral operator|=(integral) noexcept;
  integral operator^=(integral) volatile noexcept;
  integral operator^=(integral) noexcept;
};

template <class T> struct atomic<T*> {
  bool is_lock_free() const volatile noexcept;
  bool is_lock_free() const noexcept;
  void store(T*, memory_order = memory_order_seq_cst) volatile noexcept;
  void store(T*, memory_order = memory_order_seq_cst) noexcept;
  T* load(memory_order = memory_order_seq_cst) const volatile noexcept;
  T* load(memory_order = memory_order_seq_cst) const noexcept;
  operator T*() const volatile noexcept;
  operator T*() const noexcept;
  T* exchange(T*, memory_order = memory_order_seq_cst) volatile noexcept;
  T* exchange(T*, memory_order = memory_order_seq_cst) noexcept;
  bool compare_exchange_weak(T*&, T*, memory_order, memory_order) volatile noexcept;
  bool compare_exchange_weak(T*&, T*, memory_order, memory_order) noexcept;
```

```
    bool compare_exchange_strong(T*&, T*, memory_order, memory_order) volatile noexcept;
    bool compare_exchange_strong(T*&, T*, memory_order, memory_order) noexcept;
    bool compare_exchange_weak(T*&, T*, memory_order = memory_order_seq_cst) volatile noexcept;
    bool compare_exchange_weak(T*&, T*, memory_order = memory_order_seq_cst) noexcept;
    bool compare_exchange_strong(T*&, T*, memory_order = memory_order_seq_cst) volatile noexcept;
    bool compare_exchange_strong(T*&, T*, memory_order = memory_order_seq_cst) noexcept;
    T* fetch_add(ptrdiff_t, memory_order = memory_order_seq_cst) volatile noexcept;
    T* fetch_add(ptrdiff_t, memory_order = memory_order_seq_cst) noexcept;
    T* fetch_sub(ptrdiff_t, memory_order = memory_order_seq_cst) volatile noexcept;
    T* fetch_sub(ptrdiff_t, memory_order = memory_order_seq_cst) noexcept;

    atomic() noexcept = default;
    constexpr atomic(T*) noexcept;
    atomic(const atomic&) = delete;
    atomic& operator=(const atomic&) = delete;
    atomic& operator=(const atomic&) volatile = delete;
    T* operator=(T*) volatile noexcept;
    T* operator=(T*) noexcept;

    T* operator++(int) volatile noexcept;
    T* operator++(int) noexcept;
    T* operator--(int) volatile noexcept;
    T* operator--(int) noexcept;
    T* operator++() volatile noexcept;
    T* operator++() noexcept;
    T* operator--() volatile noexcept;
    T* operator--() noexcept;
    T* operator+=(ptrdiff_t) volatile noexcept;
    T* operator+=(ptrdiff_t) noexcept;
    T* operator-=(ptrdiff_t) volatile noexcept;
    T* operator-=(ptrdiff_t) noexcept;
  };
}
```

1

There is a generic class template `atomic<T>`. The type of the template argument `T` shall be trivially copyable (3.2955). [*Note:* Type arguments that are not also statically initializable may be difficult to use. — *end note*]

1

The semantics of the operations on specializations of `atomic` are defined in 29.50.

1 Specializations and instantiations of the `atomic` template shall have a deleted copy constructor, a deleted copy assignment operator, and a constexpr value constructor.

2

There shall be explicit specializations of the `atomic` template for the integral types `char`, `signed char`, `unsigned char`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `long long`, `unsigned long long`, `char16_t`, `char32_t`, `wchar_t`, and any other types needed by the typedefs in the header `<cstdint>`. For each integral type *integral*, the specialization `atomic<integral>` provides additional atomic operations appropriate to integral types. There shall be a specialization `atomic<bool>` which provides the general atomic operations as specified in 29.50.1.

1

The atomic integral specializations and the specialization `atomic<bool>` shall have standard layout. They

shall each have a trivial default constructor and a trivial destructor. They shall each support aggregate initialization syntax.

1

There shall be pointer partial specializations of the `atomic` class template. These specializations shall have standard layout, trivial default constructors, and trivial destructors. They shall each support aggregate initialization syntax.

1

There shall be named types corresponding to the integral specializations of `atomic`, as specified in Table 146, and a named type `atomic_bool` corresponding to the specified `atomic<bool>`. Each named type is either a typedef to the corresponding specialization or a base class of the corresponding specialization. If it is a base class, it shall support the same member functions as the corresponding specialization.

Table 146 — `atomic` integral typedefs

Named type	Integral argument type
`atomic_char`	`char`
`atomic_schar`	`signed char`
`atomic_uchar`	`unsigned char`
`atomic_short`	`short`
`atomic_ushort`	`unsigned short`
`atomic_int`	`int`
`atomic_uint`	`unsigned int`
`atomic_long`	`long`
`atomic_ulong`	`unsigned long`
`atomic_llong`	`long long`
`atomic_ullong`	`unsigned long long`
`atomic_char16_t`	`char16_t`
`atomic_char32_t`	`char32_t`
`atomic_wchar_t`	`wchar_t`

1 There shall be atomic typedefs corresponding to the typedefs in the header `<inttypes.h>` as specified in Table 147.

2

[*Note:* The representation of an atomic specialization need not have the same size as its corresponding argument type. Specializations should have the same size whenever possible, as this reduces the effort required to port existing code. — *end note*]

29.50 Operations on atomic types [atomics.types.operations]

29.50.1 General operations on atomic types [atomics.types.operations.general]

1 The implementation shall provide the functions and function templates identified as "general operations on atomic types" in 29.2.

2 In the declarations of these functions and function templates, the name *atomic-type* refers to either `atomic<T>` or to a named base class for `T` from Table 146 or inferred from Table 147.

29.50.2 Templated operations on atomic types [atomics.types.operations.templ]

1 The implementation shall declare but not define the function templates identified as "templated operations on atomic types" in 29.2.

Table 147 — `atomic <inttypes.h>` typedefs

Atomic typedef	<inttypes.h> type
atomic_int_least8_t	int_least8_t
atomic_uint_least8_t	uint_least8_t
atomic_int_least16_t	int_least16_t
atomic_uint_least16_t	uint_least16_t
atomic_int_least32_t	int_least32_t
atomic_uint_least32_t	uint_least32_t
atomic_int_least64_t	int_least64_t
atomic_uint_least64_t	uint_least64_t
atomic_int_fast8_t	int_fast8_t
atomic_uint_fast8_t	uint_fast8_t
atomic_int_fast16_t	int_fast16_t
atomic_uint_fast16_t	uint_fast16_t
atomic_int_fast32_t	int_fast32_t
atomic_uint_fast32_t	uint_fast32_t
atomic_int_fast64_t	int_fast64_t
atomic_uint_fast64_t	uint_fast64_t
atomic_intptr_t	intptr_t
atomic_uintptr_t	uintptr_t
atomic_size_t	size_t
atomic_ptrdiff_t	ptrdiff_t
atomic_intmax_t	intmax_t
atomic_uintmax_t	uintmax_t

29.50.3 Arithmetic operations on atomic types [atomics.types.operations.arith]

1 The implementation shall provide the functions and function template specializations identified as "arithmetic operations on atomic types" in 29.2.

2

In the declarations of these functions and function template specializations, the name *integral* refers to an integral type and the name *atomic-integral* refers to either `atomic<`*`integral`*`>` or to a named base class for *`integral`* from Table 146 or inferred from Table 147.

29.51.1 Operations on atomic pointer types [atomics.types.operations.pointer]

1 The implementation shall provide the function template specializations identified as "partial specializations for pointers" in 29.2.

29.51.2 Requirements for operations on atomic types [atomics.types.operations.req]

1 There are only a few kinds of operations on atomic types, though there are many instances on those kinds. This section specifies each general kind. The specific instances are defined in 29.32, 29.50.1, 29.50.3, and 29.51.1.

2

In the following operation definitions:

— an A refers to one of the atomic types.

— a C refers to its corresponding non-atomic type.

— an *M* refers to type of the other argument for arithmetic operations. For integral atomic types, *M* is *C*. For atomic address types, *M* is `std::ptrdiff_t`.

— the non-member functions not ending in `_explicit` have the semantics of their corresponding `_explicit` functions with `memory_order` arguments of `memory_order_seq_cst`.

1

[*Note:* Many operations are volatile-qualified. The "volatile as device register" semantics have not changed in the standard. This qualification means that volatility is preserved when applying these operations to volatile objects. It does not mean that operations on non-volatile objects become volatile. Thus, volatile qualified operations on non-volatile objects may be merged under some conditions. — *end note*]

```
A::A() noexcept = default;
```

1

Effects: leaves the atomic object in an uninitialized state. [*Note:* These semantics ensure compatibility with C. — *end note*]

```
constexpr A::A(C desired) noexcept;
```

1 *Effects:* Initializes the object with the value `desired`. Initialization is not an atomic operation (1.859). [*Note:* it is possible to have an access to an atomic object `A` race with its construction, for example by communicating the address of the just-constructed object `A` to another thread via `memory_order_relaxed` operations on a suitable atomic pointer variable, and then immediately accessing `A` in the receiving thread. This results in undefined behavior. — *end note*]

```
#define ATOMIC_VAR_INIT(value) see below
```

2

The macro expands to a token sequence suitable for constant initialization of an atomic variable of static storage duration of a type that is initialization-compatible with *value*. [*Note:* This operation may need to initialize locks. — *end note*] Concurrent access to the variable being initialized, even via an atomic operation, constitutes a data race. [*Example:*

```
atomic<int> v = ATOMIC_VAR_INIT(5);
```

— *end example*]

```
bool atomic_is_lock_free(const volatile A* object) noexcept;
bool atomic_is_lock_free(const A* object) noexcept;
bool A::is_lock_free() const volatile noexcept;
bool A::is_lock_free() const noexcept;
```

1

Returns: True if the object's operations are lock-free, false otherwise.

```
void atomic_init(volatile A* object, C desired) noexcept;
void atomic_init(A* object, C desired) noexcept;
```

1 *Effects:* Non-atomically initializes `*object` with value `desired`. This function shall only be applied to objects that have been default constructed, and then only once. [*Note:* These semantics ensure compatibility with C. — *end note*] [*Note:* Concurrent access from another thread, even via an atomic operation, constitutes a data race. — *end note*]

```
void atomic_store(volatile A* object, C desired) noexcept;
void atomic_store(A* object, C desired) noexcept;
void atomic_store_explicit(volatile A* object, C desired, memory_order order) noexcept;
void atomic_store_explicit(A* object, C desired, memory_order order) noexcept;
void A::store(C desired, memory_order order = memory_order_seq_cst) volatile noexcept;
void A::store(C desired, memory_order order = memory_order_seq_cst) noexcept;
```

2

Requires: The `order` argument shall not be `memory_order_consume`, `memory_order_acquire`, nor `memory_order_acq_rel`.

1

Effects: Atomically replaces the value pointed to by `object` or by `this` with the value of `desired`. Memory is affected according to the value of `order`.

```
C A::operator=(C desired) volatile noexcept;
C A::operator=(C desired) noexcept;
```

1 *Effects:* `store(desired)`

2 *Returns:* `desired`

```
C atomic_load(const volatile A* object) noexcept;
C atomic_load(const A* object) noexcept;
C atomic_load_explicit(const volatile A* object, memory_order) noexcept;
C atomic_load_explicit(const A* object, memory_order) noexcept;
C A::load(memory_order order = memory_order_seq_cst) const volatile noexcept;
C A::load(memory_order order = memory_order_seq_cst) const noexcept;
```

3

Requires: The `order` argument shall not be `memory_order_release` nor `memory_order_acq_rel`.

1 *Effects:* Memory is affected according to the value of `order`.

2 *Returns:* Atomically returns the value pointed to by `object` or by `this`.

```
A::operator C() const volatile noexcept;
A::operator C() const noexcept;
```

3 *Effects:* `load()`

4 *Returns:* The result of `load()`.

```
C atomic_exchange(volatile A* object, C desired) noexcept;
C atomic_exchange(A* object, C desired) noexcept;
C atomic_exchange_explicit(volatile A* object, C desired, memory_order) noexcept;
C atomic_exchange_explicit(A* object, C desired, memory_order) noexcept;
C A::exchange(C desired, memory_order order = memory_order_seq_cst) volatile noexcept;
C A::exchange(C desired, memory_order order = memory_order_seq_cst) noexcept;
```

5 *Effects:* Atomically replaces the value pointed to by `object` or by `this` with `desired`. Memory is affected according to the value of `order`. These operations are atomic read-modify-write operations (1.859).

6 *Returns:* Atomically returns the value pointed to by `object` or by `this` immediately before the effects.

```
bool atomic_compare_exchange_weak(volatile A* object, C* expected, C desired) noexcept;
bool atomic_compare_exchange_weak(A* object, C* expected, C desired) noexcept;
bool atomic_compare_exchange_strong(volatile A* object, C* expected, C desired) noexcept;
bool atomic_compare_exchange_strong(A* object, C* expected, C desired) noexcept;
bool atomic_compare_exchange_weak_explicit(volatile A* object, C* expected, C desired,
    memory_order success, memory_order failure) noexcept;
bool atomic_compare_exchange_weak_explicit(A* object, C* expected, C desired,
    memory_order success, memory_order failure) noexcept;
bool atomic_compare_exchange_strong_explicit(volatile A* object, C* expected, C desired,
    memory_order success, memory_order failure) noexcept;
bool atomic_compare_exchange_strong_explicit(A* object, C* expected, C desired,
    memory_order success, memory_order failure) noexcept;
bool A::compare_exchange_weak(C& expected, C desired,
    memory_order success, memory_order failure) volatile noexcept;
bool A::compare_exchange_weak(C& expected, C desired,
    memory_order success, memory_order failure) noexcept;
bool A::compare_exchange_strong(C& expected, C desired,
    memory_order success, memory_order failure) volatile noexcept;
bool A::compare_exchange_strong(C& expected, C desired,
    memory_order success, memory_order failure) noexcept;
bool A::compare_exchange_weak(C& expected, C desired,
    memory_order order = memory_order_seq_cst) volatile noexcept;
bool A::compare_exchange_weak(C& expected, C desired,
    memory_order order = memory_order_seq_cst) noexcept;
bool A::compare_exchange_strong(C& expected, C desired,
    memory_order order = memory_order_seq_cst) volatile noexcept;
bool A::compare_exchange_strong(C& expected, C desired,
    memory_order order = memory_order_seq_cst) noexcept;
```

7 *Requires:* The `failure` argument shall not be `memory_order_release` nor `memory_order_acq_rel`. The `failure` argument shall be no stronger than the `success` argument.

8

Effects: Atomically, compares the contents of the memory pointed to by `object` or by `this` for equality with that in `expected`, and if true, replaces the contents of the memory pointed to by `object` or by `this` with that in `desired`, and if false, updates the contents of the memory in `expected` with the contents of the memory pointed to by `object` or by `this`. Further, if the comparison is true, memory is affected according to the value of `success`, and if the comparison is false, memory is affected according to the value of `failure`. When only one `memory_order` argument is supplied, the value of `success` is `order`, and the value of `failure` is `order` except that a value of `memory_order_acq_rel` shall be replaced by the value `memory_order_acquire` and a value of `memory_order_release` shall be replaced by the value `memory_order_relaxed`. If the operation returns `true`, these operations are atomic read-modify-write operations (1.859). Otherwise, these operations are atomic load operations.

1 *Returns:* The result of the comparison.

2 [*Note:* For example, the effect of `atomic_compare_exchange_strong` is

```
if (memcmp(object, expected, sizeof(*object)) == 0)
  memcpy(object, &desired, sizeof(*object));
else
  memcpy(expected, object, sizeof(*object));
```

— *end note*] [*Example:* the expected use of the compare-and-exchange operations is as follows. The compare-and-exchange operations will update `expected` when another iteration of the loop is needed.

```
expected = current.load();
do {
  desired = function(expected);
} while (!current.compare_exchange_weak(expected, desired));
```

— *end example*]

3 Implementations should ensure that weak compare-and-exchange operations do not consistently return `false` unless either the atomic object has value different from `expected` or there are concurrent modifications to the atomic object.

1 *Remark:* A weak compare-and-exchange operation may fail spuriously. That is, even when the contents of memory referred to by `expected` and `object` are equal, it may return false and store back to `expected` the same memory contents that were originally there. [*Note:* This spurious failure enables implementation of compare-and-exchange on a broader class of machines, e.g., load-locked store-conditional machines. A consequence of spurious failure is that nearly all uses of weak compare-and-exchange will be in a loop.

When a compare-and-exchange is in a loop, the weak version will yield better performance on some platforms. When a weak compare-and-exchange would require a loop and a strong one would not, the strong one is preferable. — *end note*]

1 [*Note:* The `memcpy` and `memcmp` semantics of the compare-and-exchange operations may result in failed comparisons for values that compare equal with `operator==` if the underlying type has padding bits, trap bits, or alternate representations of the same value. Thus, `compare_exchange_strong` should be used with extreme care. On the other hand, `compare_exchange_weak` should converge rapidly. — *end note*]

1 The following operations perform arithmetic computations. The key, operator, and computation correspondence is:

Table 148 — Atomic arithmetic computations

Key	Op	Computation	Key	Op	Computation
`add`	`+`	addition	`sub`	`-`	subtraction
`or`	`|`	bitwise inclusive or	`xor`	`^`	bitwise exclusive or
`and`	`&`	bitwise and			

```
C atomic_fetch_key(volatile A* object, M operand) noexcept;
C atomic_fetch_key(A* object, M operand) noexcept;
C atomic_fetch_key_explicit(volatile A* object, M operand, memory_order order) noexcept;
C atomic_fetch_key_explicit(A* object, M operand, memory_order order) noexcept;
C A::fetch_key(M operand, memory_order order = memory_order_seq_cst) volatile noexcept;
C A::fetch_key(M operand, memory_order order = memory_order_seq_cst) noexcept;
```

2 *Effects:* Atomically replaces the value pointed to by `object` or by `this` with the result of the *computation* applied to the value pointed to by `object` or by `this` and the given `operand`. Memory is affected according to the value of `order`. These operations are atomic read-modify-write operations (1.859).

3 *Returns:* Atomically, the value pointed to by `object` or by `this` immediately before the effects.

4 *Remark:* For signed integer types, arithmetic is defined to use two's complement representation. There are no undefined results. For address types, the result may be an undefined address, but the operations otherwise have no undefined behavior.

```
C A::operator op=(M operand) volatile noexcept;
C A::operator op=(M operand) noexcept;
```

5 *Effects:* fetch_*key*(operand)

6 *Returns:* fetch_*key*(operand) op operand

```
C A::operator++(int) volatile noexcept;
C A::operator++(int) noexcept;
```

7 *Returns:* fetch_add(1)

```
C A::operator--(int) volatile noexcept;
C A::operator--(int) noexcept;
```

8 *Returns:* fetch_sub(1)

```
C A::operator++() volatile noexcept;
C A::operator++() noexcept;
```

9 *Effects:* fetch_add(1)

10 *Returns:* fetch_add(1) + 1

```
C A::operator--() volatile noexcept;
C A::operator--() noexcept;
```

11 *Effects:* fetch_sub(1)

12 *Returns:* fetch_sub(1) - 1

29.77 Flag type and operations [atomics.flag]

```
namespace std {
  typedef struct atomic_flag {
    bool test_and_set(memory_order = memory_order_seq_cst) volatile noexcept;
    bool test_and_set(memory_order = memory_order_seq_cst) noexcept;
    void clear(memory_order = memory_order_seq_cst) volatile noexcept;
    void clear(memory_order = memory_order_seq_cst) noexcept;

    atomic_flag() noexcept = default;
    atomic_flag(const atomic_flag&) = delete;
    atomic_flag& operator=(const atomic_flag&) = delete;
    atomic_flag& operator=(const atomic_flag&) volatile = delete;
  } atomic_flag;

  bool atomic_flag_test_and_set(volatile atomic_flag*) noexcept;
  bool atomic_flag_test_and_set(atomic_flag*) noexcept;
  bool atomic_flag_test_and_set_explicit(volatile atomic_flag*, memory_order) noexcept;
  bool atomic_flag_test_and_set_explicit(atomic_flag*, memory_order) noexcept;
  void atomic_flag_clear(volatile atomic_flag*) noexcept;
  void atomic_flag_clear(atomic_flag*) noexcept;
  void atomic_flag_clear_explicit(volatile atomic_flag*, memory_order) noexcept;
  void atomic_flag_clear_explicit(atomic_flag*, memory_order) noexcept;

  #define ATOMIC_FLAG_INIT see below
}
```

1 The `atomic_flag` type provides the classic test-and-set functionality. It has two states, set and clear.

2 Operations on an object of type `atomic_flag` shall be lock-free. [*Note:* Hence the operations should also be address-free. No other type requires lock-free operations, so the `atomic_flag` type is the minimum hardware-implemented type needed to conform to this International standard. The remaining types can be emulated with `atomic_flag`, though with less than ideal properties. — *end note*]

1 The `atomic_flag` type shall have standard layout. It shall have a trivial default constructor, a deleted copy constructor, a deleted copy assignment operator, and a trivial destructor.

2 The macro `ATOMIC_FLAG_INIT` shall be defined in such a way that it can be used to initialize an object of type `atomic_flag` to the clear state. The macro can be used in the form:

```
atomic_flag guard = ATOMIC_FLAG_INIT;
```

It is unspecified whether the macro can be used in other initialization contexts. For a complete static-duration object, that initialization shall be static. Unless initialized with `ATOMIC_FLAG_INIT`, it is unspecified whether an `atomic_flag` object has an initial state of set or clear.

```
bool atomic_flag_test_and_set(volatile atomic_flag* object) noexcept;
bool atomic_flag_test_and_set(atomic_flag* object) noexcept;
bool atomic_flag_test_and_set_explicit(volatile atomic_flag* object, memory_order order) noexcept;
bool atomic_flag_test_and_set_explicit(atomic_flag* object, memory_order order) noexcept;
bool atomic_flag::test_and_set(memory_order order = memory_order_seq_cst) volatile noexcept;
bool atomic_flag::test_and_set(memory_order order = memory_order_seq_cst) noexcept;
```

1

Effects: Atomically sets the value pointed to by `object` or by `this` to true. Memory is affected according to the value of `order`. These operations are atomic read-modify-write operations (1.859).

1 *Returns:* Atomically, the value of the object immediately before the effects.

```
void atomic_flag_clear(volatile atomic_flag* object) noexcept;
void atomic_flag_clear(atomic_flag* object) noexcept;
void atomic_flag_clear_explicit(volatile atomic_flag* object, memory_order order) noexcept;
void atomic_flag_clear_explicit(atomic_flag* object, memory_order order) noexcept;
void atomic_flag::clear(memory_order order = memory_order_seq_cst) volatile noexcept;
void atomic_flag::clear(memory_order order = memory_order_seq_cst) noexcept;
```

2 *Requires:* The `order` argument shall not be `memory_order_consume`, `memory_order_acquire`, nor `memory_order_acq_rel`.

3 *Effects:* Atomically sets the value pointed to by `object` or by `this` to false. Memory is affected according to the value of `order`.

29.85 Fences [atomics.fences]

1 This section introduces synchronization primitives called *fences*. Fences can have acquire semantics, release semantics, or both. A fence with acquire semantics is called an *acquire fence*. A fence with release semantics is called a *release fence*.

1 A release fence A synchronizes with an acquire fence B if there exist atomic operations X and Y, both operating on some atomic object M, such that A is sequenced before X, X modifies M, Y is sequenced before

B, and Y reads the value written by X or a value written by any side effect in the hypothetical release sequence X would head if it were a release operation.

1

A release fence A synchronizes with an atomic operation B that performs an acquire operation on an atomic object M if there exists an atomic operation X such that A is sequenced before X, X modifies M, and B reads the value written by X or a value written by any side effect in the hypothetical release sequence X would head if it were a release operation.

1 An atomic operation A that is a release operation on an atomic object M synchronizes with an acquire fence B if there exists some atomic operation X on M such that X is sequenced before B and reads the value written by A or a value written by any side effect in the release sequence headed by A.

```
extern "C" void atomic_thread_fence(memory_order order) noexcept;
```

2 *Effects:* depending on the value of `order`, this operation:

(2.1) — has no effects, if `order == memory_order_relaxed`;

(2.2) — is an acquire fence, if `order == memory_order_acquire || order == memory_order_consume`;

(2.3) — is a release fence, if `order == memory_order_release`;

(2.4) — is both an acquire fence and a release fence, if `order == memory_order_acq_rel`;

(2.5) — is a sequentially consistent acquire and release fence, if `order == memory_order_seq_cst`.

```
extern "C" void atomic_signal_fence(memory_order order) noexcept;
```

3 *Effects:* Equivalent to `atomic_thread_fence(order)`, except that the resulting ordering constraints are established only between a thread and a signal handler executed in the same thread.

4 *Note:* `atomic_signal_fence` can be used to specify the order in which actions performed by the thread become visible to the signal handler.

5 *Note:* compiler optimizations and reorderings of loads and stores are inhibited in the same way as with `atomic_thread_fence`, but the hardware fence instructions that `atomic_thread_fence` would have inserted are not emitted.

30 Thread support library [thread]

30.1 General [thread.general]

1

The following subclauses describe components to create and manage threads (1.859), perform mutual exclusion, and communicate conditions and values between threads, as summarized in Table 149.

Table 149 — Thread support library summary

	Subclause	Header(s)
30.3	Requirements	
30.25	Threads	`<thread>`
30.90	Mutual exclusion	`<mutex>` `<shared_mutex>`
30.157	Condition variables	`<condition_variable>`
30.179	Futures	`<future>`

30.3 Requirements [thread.req]

30.3.1 Template parameter names [thread.req.paramname]

1 Throughout this Clause, the names of template parameters are used to express type requirements. If a template parameter is named `Predicate`, `operator()` applied to the template argument shall return a value that is convertible to `bool`.

30.3.2 Exceptions [thread.req.exception]

1

Some functions described in this Clause are specified to throw exceptions of type `system_error` (19.24.4). Such exceptions shall be thrown if any of the function's error conditions is detected or a call to an operating system or other underlying API results in an error that prevents the library function from meeting its specifications. Failure to allocate storage shall be reported as described in 17.677.0.1.

[*Example:* Consider a function in this clause that is specified to throw exceptions of type `system_error` and specifies error conditions that include `operation_not_permitted` for a thread that does not have the privilege to perform the operation. Assume that, during the execution of this function, an `errno` of `EPERM` is reported by a POSIX API call used by the implementation. Since POSIX specifies an `errno` of `EPERM` when "the caller does not have the privilege to perform the operation", the implementation maps `EPERM` to an `error_condition` of `operation_not_permitted` (19.14) and an exception of type `system_error` is thrown. — *end example*]

1 The `error_code` reported by such an exception's `code()` member function shall compare equal to one of the conditions specified in the function's error condition element.

30.6.1 Native handles [thread.req.native]

1

Several classes described in this Clause have members `native_handle_type` and `native_handle`. The presence of these members and their semantics is implementation-defined. [*Note:* These members allow

implementations to provide access to implementation details. Their names are specified to facilitate portable compile-time detection. Actual use of these members is inherently non-portable. — *end note*]

30.8.1 Timing specifications [thread.req.timing]

1 Several functions described in this Clause take an argument to specify a timeout. These timeouts are specified as either a `duration` or a `time_point` type as specified in 20.578.

2

Implementations necessarily have some delay in returning from a timeout. Any overhead in interrupt response, function return, and scheduling induces a "quality of implementation" delay, expressed as duration D_i. Ideally, this delay would be zero. Further, any contention for processor and memory resources induces a "quality of management" delay, expressed as duration D_m. The delay durations may vary from timeout to timeout, but in all cases shorter is better.

1

The member functions whose names end in `_for` take an argument that specifies a duration. These functions produce relative timeouts. Implementations should use a steady clock to measure time for these functions.[336] Given a duration argument D_t, the real-time duration of the timeout is $D_t + D_i + D_m$.

1

The member functions whose names end in `_until` take an argument that specifies a time point. These functions produce absolute timeouts. Implementations should use the clock specified in the time point to measure time for these functions. Given a clock time point argument C_t, the clock time point of the return from timeout should be $C_t + D_i + D_m$ when the clock is not adjusted during the timeout. If the clock is adjusted to the time C_a during the timeout, the behavior should be as follows:

— if $C_a > C_t$, the waiting function should wake as soon as possible, i.e. $C_a + D_i + D_m$, since the timeout is already satisfied. [*Note:* This specification may result in the total duration of the wait decreasing when measured against a steady clock. — *end note*]

— if $C_a <= C_t$, the waiting function should not time out until `Clock::now()` returns a time $C_n >= C_t$, i.e. waking at $C_t + D_i + D_m$. [*Note:* When the clock is adjusted backwards, this specification may result in the total duration of the wait increasing when measured against a steady clock. When the clock is adjusted forwards, this specification may result in the total duration of the wait decreasing when measured against a steady clock. — *end note*]

An implementation shall return from such a timeout at any point from the time specified above to the time it would return from a steady-clock relative timeout on the difference between C_t and the time point of the call to the `_until` function. [*Note:* Implementations should decrease the duration of the wait when the clock is adjusted forwards. — *end note*]

1

[*Note:* If the clock is not synchronized with a steady clock, e.g., a CPU time clock, these timeouts might not provide useful functionality. — *end note*]

1 The resolution of timing provided by an implementation depends on both operating system and hardware. The finest resolution provided by an implementation is called the *native resolution*.

2 Implementation-provided clocks that are used for these functions shall meet the `TrivialClock` requirements (20.578.3).

3 A function that takes an argument which specifies a timeout will throw if, during its execution, a clock, time point, or time duration throws an exception. Such exceptions are referred to as *timeout-related exceptions*.

336) All implementations for which standard time units are meaningful must necessarily have a steady clock within their hardware implementation.

[*Note:* instantiations of clock, time point and duration types supplied by the implementation as specified in 20.591.2 do not throw exceptions. — *end note*]

30.16.1 Requirements for Lockable types [thread.req.lockable]

30.16.1.1 In general [thread.req.lockable.general]

1

An *execution agent* is an entity such as a thread that may perform work in parallel with other execution agents. [*Note:* Implementations or users may introduce other kinds of agents such as processes or thread-pool tasks. — *end note*] The calling agent is determined by context, e.g. the calling thread that contains the call, and so on.

1 [*Note:* Some lockable objects are "agent oblivious" in that they work for any execution agent model because they do not determine or store the agent's ID (e.g., an ordinary spin lock). — *end note*]

2 The standard library templates `unique_lock` (30.143.0.1), `lock_guard` (30.142.1.1), `lock`, `try_lock` (30.146.1), and `condition_variable_any` (30.177.1) all operate on user-supplied lockable objects. The `BasicLockable` requirements, the `Lockable` requirements, and the `TimedLockable` requirements list the requirements imposed by these library types in order to acquire or release ownership of a `lock` by a given execution agent. [*Note:* The nature of any lock ownership and any synchronization it may entail are not part of these requirements. — *end note*]

30.17.0.1 `BasicLockable` requirements [thread.req.lockable.basic]

1

A type `L` meets the `BasicLockable` requirements if the following expressions are well-formed and have the specified semantics (`m` denotes a value of type `L`).

`m.lock()`

1

Effects: Blocks until a lock can be acquired for the current execution agent. If an exception is thrown then a lock shall not have been acquired for the current execution agent.

`m.unlock()`

1 *Requires:* The current execution agent shall hold a lock on `m`.

2 *Effects:* Releases a lock on `m` held by the current execution agent.

3 *Throws:* Nothing.

30.19.0.1 `Lockable` requirements [thread.req.lockable.req]

1 A type `L` meets the `Lockable` requirements if it meets the `BasicLockable` requirements and the following expressions are well-formed and have the specified semantics (`m` denotes a value of type `L`).

`m.try_lock()`

2 *Effects:* attempts to acquire a lock for the current execution agent without blocking. If an exception is thrown then a lock shall not have been acquired for the current execution agent.

3 *Return type:* `bool`.

4

Returns: `true` if the lock was acquired, `false` otherwise.

30.21.0.1 `TimedLockable` requirements [thread.req.lockable.timed]

1 A type `L` meets the `TimedLockable` requirements if it meets the `Lockable` requirements and the following expressions are well-formed and have the specified semantics (`m` denotes a value of type `L`, `rel_time` denotes a value of an instantiation of `duration` (20.585.2), and `abs_time` denotes a value of an instantiation of `time_point` (20.591.1)).

`m.try_lock_for(rel_time)`

2 *Effects:* attempts to acquire a lock for the current execution agent within the relative timeout (30.8.1) specified by `rel_time`. The function shall not return within the timeout specified by `rel_time` unless it has obtained a lock on `m` for the current execution agent. If an exception is thrown then a lock shall not have been acquired for the current execution agent.

3 *Return type:* `bool`.

4 *Returns:* `true` if the lock was acquired, `false` otherwise.

`m.try_lock_until(abs_time)`

5 *Effects:* attempts to acquire a lock for the current execution agent before the absolute timeout (30.8.1) specified by `abs_time`. The function shall not return before the timeout specified by `abs_time` unless it has obtained a lock on `m` for the current execution agent. If an exception is thrown then a lock shall not have been acquired for the current execution agent.

6 *Return type:* `bool`.

7 *Returns:* `true` if the lock was acquired, `false` otherwise.

30.21.1 `decay_copy` [thread.decaycopy]

1

In several places in this Clause the operation *DECAY_COPY(x)* is used. All such uses mean call the function `decay_copy(x)` and use the result, where `decay_copy` is defined as follows:

```
template <class T> decay_t<T> decay_copy(T&& v)
  { return std::forward<T>(v); }
```

30.25 Threads [thread.threads]

1 30.25 describes components that can be used to create and manage threads. [*Note:* These threads are intended to map one-to-one with operating system threads. — *end note*]

Header `<thread>` synopsis

```
namespace std {
  class thread;

  void swap(thread& x, thread& y) noexcept;

  namespace this_thread {
    thread::id get_id() noexcept;

    void yield() noexcept;
    template <class Clock, class Duration>
      void sleep_until(const chrono::time_point<Clock, Duration>& abs_time);
    template <class Rep, class Period>
      void sleep_for(const chrono::duration<Rep, Period>& rel_time);
  }
}
```

30.25.1 Class `thread` [thread.thread.class]

1

The class `thread` provides a mechanism to create a new thread of execution, to join with a thread (i.e., wait for a thread to complete), and to perform other operations that manage and query the state of a thread. A `thread` object uniquely represents a particular thread of execution. That representation may be transferred to other `thread` objects in such a way that no two `thread` objects simultaneously represent the same thread of execution. A thread of execution is *detached* when no `thread` object represents that thread. Objects of class `thread` can be in a state that does not represent a thread of execution. [*Note:* A `thread` object does not represent a thread of execution after default construction, after being moved from, or after a successful call to `detach` or `join`. — *end note*]

```
namespace std {
  class thread {
  public:
    // types:
    class id;
    typedef implementation-defined native_handle_type; // See 30.6.1

    // construct/copy/destroy:
    thread() noexcept;
    template <class F, class ...Args> explicit thread(F&& f, Args&&... args);
    ~thread();
    thread(const thread&) = delete;
    thread(thread&&) noexcept;
    thread& operator=(const thread&) = delete;
    thread& operator=(thread&&) noexcept;

    // members:
    void swap(thread&) noexcept;
    bool joinable() const noexcept;
    void join();
    void detach();
    id get_id() const noexcept;
    native_handle_type native_handle(); // See 30.6.1

    // static members:
    static unsigned hardware_concurrency() noexcept;
  };
}
```

30.27.0.1 Class `thread::id` [thread.thread.id]

```
namespace std {
  class thread::id {
  public:
      id() noexcept;
  };

  bool operator==(thread::id x, thread::id y) noexcept;
  bool operator!=(thread::id x, thread::id y) noexcept;
  bool operator<(thread::id x, thread::id y) noexcept;
  bool operator<=(thread::id x, thread::id y) noexcept;
  bool operator>(thread::id x, thread::id y) noexcept;
  bool operator>=(thread::id x, thread::id y) noexcept;
```

```
  template<class charT, class traits>
    basic_ostream<charT, traits>&
      operator<< (basic_ostream<charT, traits>& out, thread::id id);

  // Hash support
  template <class T> struct hash;
  template <> struct hash<thread::id>;
}
```

1 An object of type `thread::id` provides a unique identifier for each thread of execution and a single distinct value for all `thread` objects that do not represent a thread of execution (30.25.1). Each thread of execution has an associated `thread::id` object that is not equal to the `thread::id` object of any other thread of execution and that is not equal to the `thread::id` object of any `std::thread` object that does not represent threads of execution.

1

`thread::id` shall be a trivially copyable class (Clause 9). The library may reuse the value of a `thread::id` of a terminated thread that can no longer be joined.

1 [*Note:* Relational operators allow `thread::id` objects to be used as keys in associative containers. — *end note*]

`id() noexcept;`

2 *Effects:* Constructs an object of type `id`.

3 *Postconditions:* The constructed object does not represent a thread of execution.

`bool operator==(thread::id x, thread::id y) noexcept;`

4 *Returns:* `true` only if `x` and `y` represent the same thread of execution or neither `x` nor `y` represents a thread of execution.

`bool operator!=(thread::id x, thread::id y) noexcept;`

5 *Returns:* `!(x == y)`

`bool operator<(thread::id x, thread::id y) noexcept;`

6 *Returns:* A value such that `operator<` is a total ordering as described in 25.98.

`bool operator<=(thread::id x, thread::id y) noexcept;`

7 *Returns:* `!(y < x)`

`bool operator>(thread::id x, thread::id y) noexcept;`

8 *Returns:* `y < x`

`bool operator>=(thread::id x, thread::id y) noexcept;`

9 *Returns:* `!(x < y)`

```
template<class charT, class traits>
  basic_ostream<charT, traits>&
    operator<< (basic_ostream<charT, traits>&& out, thread::id id);
```

10 *Effects:* Inserts an unspecified text representation of `id` into `out`. For two objects of type `thread::id` `x` and `y`, if `x == y` the `thread::id` objects shall have the same text representation and if `x != y` the `thread::id` objects shall have distinct text representations.

11 *Returns:* `out`

```
template <> struct hash<thread::id>;
```

12 The template specialization shall meet the requirements of class template `hash` (20.443.1).

30.31.0.1 `thread` constructors [thread.thread.constr]

```
thread() noexcept;
```

1 *Effects:* Constructs a `thread` object that does not represent a thread of execution.

2 *Postcondition:* `get_id() == id()`

```
template <class F, class ...Args> explicit thread(F&& f, Args&&... args);
```

1

Requires: `F` and each `Ti` in `Args` shall satisfy the `MoveConstructible` requirements. *`INVOKE`*`(`*`DECAY_COPY`*`( std::forward<F>(f)), `*`DECAY_COPY`*`(std::forward<Args>(args))...)` (20.295.1) shall be a valid expression.

1 *Remarks:* This constructor shall not participate in overload resolution if `decay_t<F>` is the same type as `std::thread`.

2

Effects: Constructs an object of type `thread`. The new thread of execution executes *`INVOKE`*`(`*`DECAY_COPY`*`( std::forward<F>(f)), `*`DECAY_COPY`*`(std::forward<Args>(args))...)` with the calls to *`DECAY_COPY`* being evaluated in the constructing thread. Any return value from this invocation is ignored. [*Note:* This implies that any exceptions not thrown from the invocation of the copy of `f` will be thrown in the constructing thread, not the new thread. — *end note*] If the invocation of *`INVOKE`*`(`*`DECAY_COPY`*`( std::forward<F>(f)), `*`DECAY_COPY`*`(std::forward<Args>(args))...)` terminates with an uncaught exception, `std::terminate` shall be called.

1 *Synchronization:* The completion of the invocation of the constructor synchronizes with the beginning of the invocation of the copy of `f`.

1 *Postconditions:* `get_id() != id()`. `*this` represents the newly started thread.

2 *Throws:* `system_error` if unable to start the new thread.

3 *Error conditions:*

— `resource_unavailable_try_again` — the system lacked the necessary resources to create another thread, or the system-imposed limit on the number of threads in a process would be exceeded.

```
thread(thread&& x) noexcept;
```

1 *Effects:* Constructs an object of type `thread` from `x`, and sets `x` to a default constructed state.

2 *Postconditions:* `x.get_id() == id()` and `get_id()` returns the value of `x.get_id()` prior to the start of construction.

30.60.0.1 `thread` destructor [thread.thread.destr]

```
~thread();
```

1

If `joinable()`, calls `std::terminate()`. Otherwise, has no effects. [*Note:* Either implicitly detaching or joining a `joinable()` thread in its destructor could result in difficult to debug correctness (for detach) or performance (for join) bugs encountered only when an exception is raised. Thus the programmer must ensure that the destructor is never executed while the thread is still joinable. — *end note*]

30.74.0.1 `thread` assignment [thread.thread.assign]

```
thread& operator=(thread&& x) noexcept;
```

1

Effects: If `joinable()`, calls `std::terminate()`. Otherwise, assigns the state of `x` to `*this` and sets `x` to a default constructed state.

1 *Postconditions:* `x.get_id() == id()` and `get_id()` returns the value of `x.get_id()` prior to the assignment.

2 *Returns:* `*this`

30.75.0.1 `thread` members [thread.thread.member]

```
void swap(thread& x) noexcept;
```

1 *Effects:* Swaps the state of `*this` and `x`.

```
bool joinable() const noexcept;
```

2 *Returns:* `get_id() != id()`

```
void join();
```

3 *Requires:* `joinable()` is `true`.

4

Effects: Blocks until the thread represented by `*this` has completed.

1

Synchronization: The completion of the thread represented by `*this` synchronizes with (1.859) the corresponding successful `join()` return. [*Note:* Operations on `*this` are not synchronized. — *end note*]

1 *Postconditions:* The thread represented by `*this` has completed. `get_id() == id()`.

2

Throws: `system_error` when an exception is required (30.3.2).

1 *Error conditions:*

(1.1) — `resource_deadlock_would_occur` — if deadlock is detected or `this->get_id() == std::this_thread::get_id()`.

(1.2) — `no_such_process` — if the thread is not valid.

(1.3) — `invalid_argument` — if the thread is not joinable.

```
void detach();
```

2 *Requires:* `joinable()` is `true`.

3

Effects: The thread represented by `*this` continues execution without the calling thread blocking. When `detach()` returns, `*this` no longer represents the possibly continuing thread of execution. When the thread previously represented by `*this` ends execution, the implementation shall release any owned resources.

1 *Postcondition:* `get_id() == id()`.

2 *Throws:* `system_error` when an exception is required (30.3.2).

3 *Error conditions:*

(3.1) — `no_such_process` — if the thread is not valid.

(3.2) — `invalid_argument` — if the thread is not joinable.

```
id get_id() const noexcept;
```

4 *Returns:* A default constructed `id` object if `*this` does not represent a thread, otherwise `this_thread::get_id()` for the thread of execution represented by `*this`.

30.85.0.1 `thread` static members [thread.thread.static]

```
unsigned hardware_concurrency() noexcept;
```

1 *Returns:* The number of hardware thread contexts. [*Note:* This value should only be considered to be a hint. — *end note*] If this value is not computable or well defined an implementation should return 0.

30.85.0.2 `thread` specialized algorithms [thread.thread.algorithm]

```
void swap(thread& x, thread& y) noexcept;
```

1 *Effects:* `x.swap(y)`

30.85.1 Namespace `this_thread` [thread.thread.this]

```
namespace std {
  namespace this_thread {
    thread::id get_id() noexcept;

    void yield() noexcept;
    template <class Clock, class Duration>
      void sleep_until(const chrono::time_point<Clock, Duration>& abs_time);
    template <class Rep, class Period>
      void sleep_for(const chrono::duration<Rep, Period>& rel_time);
  }
}
```

```
thread::id this_thread::get_id() noexcept;
```

1 *Returns:* An object of type `thread::id` that uniquely identifies the current thread of execution. No other thread of execution shall have this id and this thread of execution shall always have this id. The object returned shall not compare equal to a default constructed `thread::id`.

```
void this_thread::yield() noexcept;
```

2

Effects: Offers the implementation the opportunity to reschedule.

1 *Synchronization:* None.

```
template <class Clock, class Duration>
  void sleep_until(const chrono::time_point<Clock, Duration>& abs_time);
```

2 *Effects:* Blocks the calling thread for the absolute timeout (30.8.1) specified by `abs_time`.

3 *Synchronization:* None.

4 *Throws:* Timeout-related exceptions (30.8.1).

```
template <class Rep, class Period>
  void sleep_for(const chrono::duration<Rep, Period>& rel_time);
```

5

Effects: Blocks the calling thread for the relative timeout (30.8.1) specified by `rel_time`.

1 *Synchronization:* None.

2 *Throws:* Timeout-related exceptions (30.8.1).

30.90 Mutual exclusion [thread.mutex]

1 This section provides mechanisms for mutual exclusion: mutexes, locks, and call once. These mechanisms ease the production of race-free programs (1.859).

Header `<mutex>` synopsis

```
namespace std {
  class mutex;
  class recursive_mutex;
  class timed_mutex;
  class recursive_timed_mutex;

  struct defer_lock_t { };
  struct try_to_lock_t { };
  struct adopt_lock_t { };

  constexpr defer_lock_t  defer_lock { };
  constexpr try_to_lock_t try_to_lock { };
  constexpr adopt_lock_t  adopt_lock { };

  template <class Mutex> class lock_guard;
  template <class Mutex> class unique_lock;

  template <class Mutex>
    void swap(unique_lock<Mutex>& x, unique_lock<Mutex>& y) noexcept;

  template <class L1, class L2, class... L3> int try_lock(L1&, L2&, L3&...);
  template <class L1, class L2, class... L3> void lock(L1&, L2&, L3&...);

  struct once_flag {
    constexpr once_flag() noexcept;

    once_flag(const once_flag&) = delete;
    once_flag& operator=(const once_flag&) = delete;
  };

  template<class Callable, class ...Args>
    void call_once(once_flag& flag, Callable&& func, Args&&... args);
}
```

Header `<shared_mutex>` synopsis

```
namespace std {
  class shared_timed_mutex;
  template <class Mutex> class shared_lock;
  template <class Mutex>
    void swap(shared_lock<Mutex>& x, shared_lock<Mutex>& y) noexcept;
}
```

30.90.1 Mutex requirements [thread.mutex.requirements]

30.90.1.1 In general [thread.mutex.requirements.general]

1

A mutex object facilitates protection against data races and allows safe synchronization of data between execution agents (30.16.1). An execution agent *owns* a mutex from the time it successfully calls one of the lock functions until it calls unlock. Mutexes can be either recursive or non-recursive, and can grant simultaneous ownership to one or many execution agents. Both recursive and non-recursive mutexes are supplied.

30.92.0.1 Mutex types [thread.mutex.requirements.mutex]

1 The *mutex types* are the standard library types `std::mutex`, `std::recursive_mutex`, `std::timed_mutex`, `std::recursive_timed_mutex`, and `std::shared_timed_mutex`. They shall meet the requirements set out in this section. In this description, `m` denotes an object of a mutex type.

2

The mutex types shall meet the `Lockable` requirements (30.19.0.1).

1

The mutex types shall be `DefaultConstructible` and `Destructible`. If initialization of an object of a mutex type fails, an exception of type `system_error` shall be thrown. The mutex types shall not be copyable or movable.

1

The error conditions for error codes, if any, reported by member functions of the mutex types shall be:

— `resource_unavailable_try_again` — if any native handle type manipulated is not available.

— `operation_not_permitted` — if the thread does not have the privilege to perform the operation.

— `device_or_resource_busy` — if any native handle type manipulated is already locked.

— `invalid_argument` — if any native handle type manipulated as part of mutex construction is incorrect.

1 The implementation shall provide lock and unlock operations, as described below. For purposes of determining the existence of a data race, these behave as atomic operations (1.859). The lock and unlock operations on a single mutex shall appear to occur in a single total order. [*Note:* this can be viewed as the modification order (1.859) of the mutex. — *end note*] [*Note:* Construction and destruction of an object of a mutex type need not be thread-safe; other synchronization should be used to ensure that mutex objects are initialized and visible to other threads. — *end note*]

2

The expression `m.lock()` shall be well-formed and have the following semantics:

1

Requires: If `m` is of type `std::mutex`, `std::timed_mutex`, or `std::shared_timed_mutex`, the calling thread does not own the mutex.

1 *Effects:* Blocks the calling thread until ownership of the mutex can be obtained for the calling thread.

2 *Postcondition:* The calling thread owns the mutex.

3 *Return type:* `void`

4 *Synchronization:* Prior `unlock()` operations on the same object shall *synchronize with* (1.859) this operation.

5 *Throws:* `system_error` when an exception is required (30.3.2).

6 *Error conditions:*

— `operation_not_permitted` — if the thread does not have the privilege to perform the operation.

— `resource_deadlock_would_occur` — if the implementation detects that a deadlock would occur.

— `device_or_resource_busy` — if the mutex is already locked and blocking is not possible.

1 The expression `m.try_lock()` shall be well-formed and have the following semantics:

2 *Requires:* If `m` is of type `std::mutex`, `std::timed_mutex`, or `std::shared_timed_mutex`, the calling thread does not own the mutex.

3 *Effects:* Attempts to obtain ownership of the mutex for the calling thread without blocking. If ownership is not obtained, there is no effect and `try_lock()` immediately returns. An implementation may fail to obtain the lock even if it is not held by any other thread. [*Note:* This spurious failure is normally uncommon, but allows interesting implementations based on a simple compare and exchange (Clause 29). — *end note*] An implementation should ensure that `try_lock()` does not consistently return `false` in the absence of contending mutex acquisitions.

1 *Return type:* `bool`

2 *Returns:* `true` if ownership of the mutex was obtained for the calling thread, otherwise `false`.

3 *Synchronization:* If `try_lock()` returns `true`, prior `unlock()` operations on the same object *synchronize with* (1.859) this operation. [*Note:* Since `lock()` does not synchronize with a failed subsequent `try_lock()`, the visibility rules are weak enough that little would be known about the state after a failure, even in the absence of spurious failures. — *end note*]

1 *Throws:* Nothing.

2

The expression `m.unlock()` shall be well-formed and have the following semantics:

1 *Requires:* The calling thread shall own the mutex.

2 *Effects:* Releases the calling thread's ownership of the mutex.

1 *Return type:* `void`

2 *Synchronization:* This operation synchronizes with (1.859) subsequent lock operations that obtain ownership on the same object.

1 *Throws:* Nothing.

30.125.0.0.1 Class `mutex` [thread.mutex.class]

```
namespace std {
  class mutex {
  public:
    constexpr mutex() noexcept;
    ~mutex();

    mutex(const mutex&) = delete;
    mutex& operator=(const mutex&) = delete;
```

```
    void lock();
    bool try_lock();
    void unlock();

    typedef implementation-defined native_handle_type; // See 30.6.1
    native_handle_type native_handle();                // See 30.6.1
  };
}
```

1

The class `mutex` provides a non-recursive mutex with exclusive ownership semantics. If one thread owns a mutex object, attempts by another thread to acquire ownership of that object will fail (for `try_lock()`) or block (for `lock()`) until the owning thread has released ownership with a call to `unlock()`.

1 [*Note:* After a thread `A` has called `unlock()`, releasing a mutex, it is possible for another thread `B` to lock the same mutex, observe that it is no longer in use, unlock it, and destroy it, before thread `A` appears to have returned from its unlock call. Implementations are required to handle such scenarios correctly, as long as thread `A` doesn't access the mutex after the unlock call returns. These cases typically occur when a reference-counted object contains a mutex that is used to protect the reference count. — *end note*]

2 The class `mutex` shall satisfy all the `Mutex` requirements (30.90.1). It shall be a standard-layout class (Clause 9).

3

[*Note:* A program may deadlock if the thread that owns a `mutex` object calls `lock()` on that object. If the implementation can detect the deadlock, a `resource_deadlock_would_occur` error condition may be observed. — *end note*]

1

The behavior of a program is undefined if it destroys a `mutex` object owned by any thread or a thread terminates while owning a `mutex` object.

30.134.0.0.1 Class `recursive_mutex` [thread.mutex.recursive]

```
namespace std {
  class recursive_mutex {
  public:
    recursive_mutex();
    ~recursive_mutex();

    recursive_mutex(const recursive_mutex&) = delete;
    recursive_mutex& operator=(const recursive_mutex&) = delete;

    void lock();
    bool try_lock() noexcept;
    void unlock();

    typedef implementation-defined native_handle_type; // See 30.6.1
    native_handle_type native_handle();                // See 30.6.1
  };
}
```

1 The class `recursive_mutex` provides a recursive mutex with exclusive ownership semantics. If one thread owns a `recursive_mutex` object, attempts by another thread to acquire ownership of that object will fail (for `try_lock()`) or block (for `lock()`) until the first thread has completely released ownership.

2 The class `recursive_mutex` shall satisfy all the Mutex requirements (30.90.1). It shall be a standard-layout class (Clause 9).

3

A thread that owns a `recursive_mutex` object may acquire additional levels of ownership by calling `lock()` or `try_lock()` on that object. It is unspecified how many levels of ownership may be acquired by a single thread. If a thread has already acquired the maximum level of ownership for a `recursive_mutex` object, additional calls to `try_lock()` shall fail, and additional calls to `lock()` shall throw an exception of type `system_error`. A thread shall call `unlock()` once for each level of ownership acquired by calls to `lock()` and `try_lock()`. Only when all levels of ownership have been released may ownership be acquired by another thread.

1 The behavior of a program is undefined if:

(1.1) — it destroys a `recursive_mutex` object owned by any thread or

(1.2) — a thread terminates while owning a `recursive_mutex` object.

30.135.0.1 Timed mutex types [thread.timedmutex.requirements]

1 The *timed mutex types* are the standard library types `std::timed_mutex`, `std::recursive_timed_mutex`, and `std::shared_timed_mutex`. They shall meet the requirements set out below. In this description, `m` denotes an object of a mutex type, `rel_time` denotes an object of an instantiation of `duration` (20.585.2), and `abs_time` denotes an object of an instantiation of `time_point` (20.591.1).

2 The timed mutex types shall meet the `TimedLockable` requirements (30.21.0.1).

3 The expression `m.try_lock_for(rel_time)` shall be well-formed and have the following semantics:

4 *Requires:* If `m` is of type `std::timed_mutex` or `std::shared_timed_mutex`, the calling thread does not own the mutex.

5 *Effects:* The function attempts to obtain ownership of the mutex within the relative timeout (30.8.1) specified by `rel_time`. If the time specified by `rel_time` is less than or equal to `rel_time.zero()`, the function attempts to obtain ownership without blocking (as if by calling `try_lock()`). The function shall return within the timeout specified by `rel_time` only if it has obtained ownership of the mutex object. [*Note:* As with `try_lock()`, there is no guarantee that ownership will be obtained if the lock is available, but implementations are expected to make a strong effort to do so. — *end note*]

6 *Return type:* `bool`

7 *Returns:* `true` if ownership was obtained, otherwise `false`.

8 *Synchronization:* If `try_lock_for()` returns `true`, prior `unlock()` operations on the same object *synchronize with* (1.859) this operation.

9 *Throws:* Timeout-related exceptions (30.8.1).

10 The expression `m.try_lock_until(abs_time)` shall be well-formed and have the following semantics:

11 *Requires:* If `m` is of type `std::timed_mutex` or `std::shared_timed_mutex`, the calling thread does not own the mutex.

12 *Effects:* The function attempts to obtain ownership of the mutex. If `abs_time` has already passed, the function attempts to obtain ownership without blocking (as if by calling `try_lock()`). The function shall return before the absolute timeout (30.8.1) specified by `abs_time` only if it has obtained ownership of the mutex object. [*Note:* As with `try_lock()`, there is no guarantee that ownership will be obtained if the lock is available, but implementations are expected to make a strong effort to do so. — *end note*]

13 *Return type:* `bool`

14 *Returns:* `true` if ownership was obtained, otherwise `false`.

15 *Synchronization:* If `try_lock_until()` returns `true`, prior `unlock()` operations on the same object *synchronize with* (1.859) this operation.

16 *Throws:* Timeout-related exceptions (30.8.1).

30.135.0.1.1 Class `timed_mutex` [thread.timedmutex.class]

```
namespace std {
  class timed_mutex {
  public:
    timed_mutex();
    ~timed_mutex();

    timed_mutex(const timed_mutex&) = delete;
    timed_mutex& operator=(const timed_mutex&) = delete;

    void lock();  // blocking
    bool try_lock();
    template <class Rep, class Period>
      bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);
    template <class Clock, class Duration>
      bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);
    void unlock();

    typedef implementation-defined native_handle_type; // See 30.6.1
    native_handle_type native_handle();                // See 30.6.1
  };
}
```

1 The class `timed_mutex` provides a non-recursive mutex with exclusive ownership semantics. If one thread owns a `timed_mutex` object, attempts by another thread to acquire ownership of that object will fail (for `try_lock()`) or block (for `lock()`, `try_lock_for()`, and `try_lock_until()`) until the owning thread has released ownership with a call to `unlock()` or the call to `try_lock_for()` or `try_lock_until()` times out (having failed to obtain ownership).

2 The class `timed_mutex` shall satisfy all of the `TimedMutex` requirements (30.135.0.1). It shall be a standard-layout class (Clause 9).

3 The behavior of a program is undefined if:

(3.1) — it destroys a `timed_mutex` object owned by any thread,

(3.2) — a thread that owns a `timed_mutex` object calls `lock()`, `try_lock()`, `try_lock_for()`, or `try_lock_until()` on that object, or

(3.3) — a thread terminates while owning a `timed_mutex` object.

30.135.0.1.2 Class `recursive_timed_mutex` [thread.timedmutex.recursive]

```
namespace std {
  class recursive_timed_mutex {
  public:
    recursive_timed_mutex();
    ~recursive_timed_mutex();

    recursive_timed_mutex(const recursive_timed_mutex&) = delete;
    recursive_timed_mutex& operator=(const recursive_timed_mutex&) = delete;
```

```
    void lock();  // blocking
    bool try_lock() noexcept;
    template <class Rep, class Period>
      bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);
    template <class Clock, class Duration>
      bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);
    void unlock();

    typedef implementation-defined native_handle_type; // See 30.6.1
    native_handle_type native_handle();                // See 30.6.1
  };
}
```

1 The class `recursive_timed_mutex` provides a recursive mutex with exclusive ownership semantics. If one thread owns a `recursive_timed_mutex` object, attempts by another thread to acquire ownership of that object will fail (for `try_lock()`) or block (for `lock()`, `try_lock_for()`, and `try_lock_until()`) until the owning thread has completely released ownership or the call to `try_lock_for()` or `try_lock_until()` times out (having failed to obtain ownership).

2 The class `recursive_timed_mutex` shall satisfy all of the `TimedMutex` requirements (30.135.0.1). It shall be a standard-layout class (Clause 9).

3 A thread that owns a `recursive_timed_mutex` object may acquire additional levels of ownership by calling `lock()`, `try_lock()`, `try_lock_for()`, or `try_lock_until()` on that object. It is unspecified how many levels of ownership may be acquired by a single thread. If a thread has already acquired the maximum level of ownership for a `recursive_timed_mutex` object, additional calls to `try_lock()`, `try_lock_for()`, or `try_lock_until()` shall fail, and additional calls to `lock()` shall throw an exception of type `system_error`. A thread shall call `unlock()` once for each level of ownership acquired by calls to `lock()`, `try_lock()`, `try_lock_for()`, and `try_lock_until()`. Only when all levels of ownership have been released may ownership of the object be acquired by another thread.

4 The behavior of a program is undefined if:

(4.1) — it destroys a `recursive_timed_mutex` object owned by any thread, or

(4.2) — a thread terminates while owning a `recursive_timed_mutex` object.

30.135.0.2 Shared timed mutex types [thread.sharedtimedmutex.requirements]

1 The standard library type `std::shared_timed_mutex` is a *shared timed mutex type*. Shared timed mutex types shall meet the requirements of timed mutex types (30.135.0.1), and additionally shall meet the requirements set out below. In this description, `m` denotes an object of a mutex type, `rel_type` denotes an object of an instantiation of `duration` (20.585.2), and `abs_time` denotes an object of an instantiation of `time_point` (20.591.1).

2 In addition to the exclusive lock ownership mode specified in 30.92.0.1, shared mutex types provide a *shared lock* ownership mode. Multiple execution agents can simultaneously hold a shared lock ownership of a shared mutex type. But no execution agent shall hold a shared lock while another execution agent holds an exclusive lock on the same shared mutex type, and vice-versa. The maximum number of execution agents which can share a shared lock on a single shared mutex type is unspecified, but shall be at least 10000. If more than the maximum number of execution agents attempt to obtain a shared lock, the excess execution agents shall block until the number of shared locks are reduced below the maximum amount by other execution agents releasing their shared lock.

3 The expression `m.lock_shared()` shall be well-formed and have the following semantics:

4 *Requires:* The calling thread has no ownership of the mutex.

5

Effects: Blocks the calling thread until shared ownership of the mutex can be obtained for the calling thread. If an exception is thrown then a shared lock shall not have been acquired for the current thread.

1 *Postcondition:* The calling thread has a shared lock on the mutex.

2 *Return type:* `void`.

3

Synchronization: Prior `unlock()` operations on the same object shall synchronize with (1.859) this operation.

1 *Throws:* `system_error` when an exception is required (30.3.2).

2 *Error conditions:*

(2.1) — `operation_not_permitted` — if the thread does not have the privilege to perform the operation.

(2.2) — `resource_deadlock_would_occur` — if the implementation detects that a deadlock would occur.

(2.3) — `device_or_resource_busy` — if the mutex is already locked and blocking is not possible.

3 The expression `m.unlock_shared()` shall be well-formed and have the following semantics:

4 *Requires:* The calling thread shall hold a shared lock on the mutex.

5 *Effects:* Releases a shared lock on the mutex held by the calling thread.

6 *Return type:* `void`.

7 *Synchronization:* This operation synchronizes with (1.859) subsequent `lock()` operations that obtain ownership on the same object.

8 *Throws:* Nothing.

9 The expression `m.try_lock_shared()` shall be well-formed and have the following semantics:

10 *Requires:* The calling thread has no ownership of the mutex.

11 *Effects:* Attempts to obtain shared ownership of the mutex for the calling thread without blocking. If shared ownership is not obtained, there is no effect and `try_lock_shared()` immediately returns. An implementation may fail to obtain the lock even if it is not held by any other thread.

12 *Return type:* `bool`.

13 *Returns:* `true` if the shared ownership lock was acquired, `false` otherwise.

14 *Synchronization:* If `try_lock_shared()` returns `true`, prior `unlock()` operations on the same object synchronize with (1.859) this operation.

15 *Throws:* Nothing.

16 The expression `m.try_lock_shared_for(rel_time)` shall be well-formed and have the following semantics:

17 *Requires:* The calling thread has no ownership of the mutex.

18 *Effects:* Attempts to obtain shared lock ownership for the calling thread within the relative timeout (30.8.1) specified by `rel_time`. If the time specified by `rel_time` is less than or equal to `rel_time.zero()`, the function attempts to obtain ownership without blocking (as if by calling `try_lock_shared()`). The function shall return within the timeout specified by `rel_time` only if it has obtained shared ownership of the mutex object. [*Note:* As with `try_lock()`, there is no guarantee that ownership will be obtained if the lock is available, but implementations are expected to make a strong effort to do so. — *end note*] If an exception is thrown then a shared lock shall not have been acquired for the current thread.

19 *Return type:* `bool`.

20 *Returns:* `true` if the shared lock was acquired, `false` otherwise.

21 *Synchronization:* If `try_lock_shared_for()` returns `true`, prior `unlock()` operations on the same object synchronize with (1.859) this operation.

22 *Throws:* Timeout-related exceptions (30.8.1).

23 The expression `m.try_lock_shared_until(abs_time)` shall be well-formed and have the following semantics:

24 *Requires:* The calling thread has no ownership of the mutex.

25 *Effects:* The function attempts to obtain shared ownership of the mutex. If `abs_time` has already passed, the function attempts to obtain shared ownership without blocking (as if by calling `try_lock_shared()`). The function shall return before the absolute timeout (30.8.1) specified by `abs_time` only if it has obtained shared ownership of the mutex object. [*Note:* As with `try_lock()`, there is no guarantee that ownership will be obtained if the lock is available, but implementations are expected to make a strong effort to do so. — *end note*] If an exception is thrown then a shared lock shall not have been acquired for the current thread.

26 *Return type:* `bool`.

27 *Returns:* `true` if the shared lock was acquired, `false` otherwise.

28 *Synchronization:* If `try_lock_shared_until()` returns `true`, prior `unlock()` operations on the same object synchronize with (1.859) this operation.

29 *Throws:* Timeout-related exceptions (30.8.1).

30.142.0.0.1 Class `shared_timed_mutex` [thread.sharedtimedmutex.class]

```
namespace std {
  class shared_timed_mutex {
  public:
    shared_timed_mutex();
    ~shared_timed_mutex();

    shared_timed_mutex(const shared_timed_mutex&) = delete;
    shared_timed_mutex& operator=(const shared_timed_mutex&) = delete;

    // Exclusive ownership
    void lock();  // blocking
    bool try_lock();
    template <class Rep, class Period>
      bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);
    template <class Clock, class Duration>
      bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);
    void unlock();

    // Shared ownership
    void lock_shared();  // blocking
    bool try_lock_shared();
    template <class Rep, class Period>
      bool
      try_lock_shared_for(const chrono::duration<Rep, Period>& rel_time);
    template <class Clock, class Duration>
      bool
      try_lock_shared_until(const chrono::time_point<Clock, Duration>& abs_time);
```

```
    void unlock_shared();
  };
}
```

1 The class `shared_timed_mutex` provides a non-recursive mutex with shared ownership semantics.

2 The class `shared_timed_mutex` shall satisfy all of the `SharedTimedMutex` requirements (30.135.0.2). It shall be a standard-layout class (Clause 9).

3 The behavior of a program is undefined if:

(3.1) — it destroys a `shared_timed_mutex` object owned by any thread,

(3.2) — a thread attempts to recursively gain any ownership of a `shared_timed_mutex`, or

(3.3) — a thread terminates while possessing any ownership of a `shared_timed_mutex`.

30.142.1 Locks [thread.lock]

1 A *lock* is an object that holds a reference to a lockable object and may unlock the lockable object during the lock's destruction (such as when leaving block scope). An execution agent may use a lock to aid in managing ownership of a lockable object in an exception safe manner. A lock is said to *own* a lockable object if it is currently managing the ownership of that lockable object for an execution agent. A lock does not manage the lifetime of the lockable object it references. [*Note:* Locks are intended to ease the burden of unlocking the lockable object under both normal and exceptional circumstances. — *end note*]

2 Some lock constructors take tag types which describe what should be done with the lockable object during the lock's construction.

```
namespace std {
  struct defer_lock_t  { };     // do not acquire ownership of the mutex
  struct try_to_lock_t { };     // try to acquire ownership of the mutex
                                // without blocking
  struct adopt_lock_t  { };     // assume the calling thread has already
                                // obtained mutex ownership and manage it

  constexpr defer_lock_t   defer_lock { };
  constexpr try_to_lock_t  try_to_lock { };
  constexpr adopt_lock_t   adopt_lock { };
}
```

30.142.1.1 Class template `lock_guard` [thread.lock.guard]

```
namespace std {
  template <class Mutex>
  class lock_guard {
  public:
    typedef Mutex mutex_type;

    explicit lock_guard(mutex_type& m);
    lock_guard(mutex_type& m, adopt_lock_t);
    ~lock_guard();

    lock_guard(lock_guard const&) = delete;
    lock_guard& operator=(lock_guard const&) = delete;

  private:
    mutex_type& pm; // exposition only
```

```
  };
}
```

1

An object of type `lock_guard` controls the ownership of a lockable object within a scope. A `lock_guard` object maintains ownership of a lockable object throughout the `lock_guard` object's lifetime (3.2563). The behavior of a program is undefined if the lockable object referenced by `pm` does not exist for the entire lifetime of the `lock_guard` object. The supplied `Mutex` type shall meet the `BasicLockable` requirements (30.17.0.1).

`explicit lock_guard(mutex_type& m);`

1 *Requires:* If `mutex_type` is not a recursive mutex, the calling thread does not own the mutex `m`.

2 *Effects:* `m.lock()`

3 *Postcondition:* `&pm == &m`

`lock_guard(mutex_type& m, adopt_lock_t);`

4 *Requires:* The calling thread owns the mutex `m`.

5 *Postcondition:* `&pm == &m`

6 *Throws:* Nothing.

`~lock_guard();`

7 *Effects:* `pm.unlock()`

30.143.0.1 Class template `unique_lock` [thread.lock.unique]

```
namespace std {
  template <class Mutex>
  class unique_lock {
  public:
    typedef Mutex mutex_type;

    // 30.145.0.0.1, construct/copy/destroy:
    unique_lock() noexcept;
    explicit unique_lock(mutex_type& m);
    unique_lock(mutex_type& m, defer_lock_t) noexcept;
    unique_lock(mutex_type& m, try_to_lock_t);
    unique_lock(mutex_type& m, adopt_lock_t);
    template <class Clock, class Duration>
      unique_lock(mutex_type& m, const chrono::time_point<Clock, Duration>& abs_time);
    template <class Rep, class Period>
      unique_lock(mutex_type& m, const chrono::duration<Rep, Period>& rel_time);
    ~unique_lock();

    unique_lock(unique_lock const&) = delete;
    unique_lock& operator=(unique_lock const&) = delete;

    unique_lock(unique_lock&& u) noexcept;
    unique_lock& operator=(unique_lock&& u);

    // 30.146.0.0.1, locking:
    void lock();
    bool try_lock();
```

```
    template <class Rep, class Period>
      bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);
    template <class Clock, class Duration>
      bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);

    void unlock();

    // 30.146.0.0.2, modifiers:
    void swap(unique_lock& u) noexcept;
    mutex_type* release() noexcept;

    // 30.146.0.0.3, observers:
    bool owns_lock() const noexcept;
    explicit operator bool () const noexcept;
    mutex_type* mutex() const noexcept;

  private:
    mutex_type* pm; // exposition only
    bool owns;      // exposition only
  };

  template <class Mutex>
    void swap(unique_lock<Mutex>& x, unique_lock<Mutex>& y) noexcept;
}
```

1

An object of type `unique_lock` controls the ownership of a lockable object within a scope. Ownership of the lockable object may be acquired at construction or after construction, and may be transferred, after acquisition, to another `unique_lock` object. Objects of type `unique_lock` are not copyable but are movable. The behavior of a program is undefined if the contained pointer `pm` is not null and the lockable object pointed to by `pm` does not exist for the entire remaining lifetime (3.2563) of the `unique_lock` object. The supplied `Mutex` type shall meet the `BasicLockable` requirements (30.17.0.1).

1 [*Note:* `unique_lock<Mutex>` meets the `BasicLockable` requirements. If `Mutex` meets the `Lockable` requirements (30.19.0.1), `unique_lock<Mutex>` also meets the `Lockable` requirements; if `Mutex` meets the `TimedLockable` requirements (30.21.0.1), `unique_lock<Mutex>` also meets the `TimedLockable` requirements. — *end note*]

30.145.0.0.1 `unique_lock` constructors, destructor, and assignment [thread.lock.unique.cons]

`unique_lock() noexcept;`

1 *Effects:* Constructs an object of type `unique_lock`.

2 *Postconditions:* `pm == 0` and `owns == false`.

`explicit unique_lock(mutex_type& m);`

3 *Requires:* If `mutex_type` is not a recursive mutex the calling thread does not own the mutex.

4 *Effects:* Constructs an object of type `unique_lock` and calls `m.lock()`.

5 *Postconditions:* `pm == &m` and `owns == true`.

`unique_lock(mutex_type& m, defer_lock_t) noexcept;`

6 *Effects:* Constructs an object of type `unique_lock`.

7 *Postconditions:* `pm == &m` and `owns == false`.

```
unique_lock(mutex_type& m, try_to_lock_t);
```

8

Requires: The supplied `Mutex` type shall meet the `Lockable` requirements (30.19.0.1). If `mutex_type` is not a recursive mutex the calling thread does not own the mutex.

1 *Effects:* Constructs an object of type `unique_lock` and calls `m.try_lock()`.

2 *Postconditions:* `pm == &m` and `owns == res`, where `res` is the value returned by the call to `m.try_lock()`.

```
unique_lock(mutex_type& m, adopt_lock_t);
```

3 *Requires:* The calling thread own the mutex.

4 *Effects:* Constructs an object of type `unique_lock`.

5 *Postconditions:* `pm == &m` and `owns == true`.

6 *Throws:* Nothing.

```
template <class Clock, class Duration>
  unique_lock(mutex_type& m, const chrono::time_point<Clock, Duration>& abs_time);
```

7 *Requires:* If `mutex_type` is not a recursive mutex the calling thread does not own the mutex. The supplied `Mutex` type shall meet the `TimedLockable` requirements (30.21.0.1).

8 *Effects:* Constructs an object of type `unique_lock` and calls `m.try_lock_until(abs_time)`.

9 *Postconditions:* `pm == &m` and `owns == res`, where `res` is the value returned by the call to `m.try_lock_until(abs_time)`.

```
template <class Rep, class Period>
  unique_lock(mutex_type& m, const chrono::duration<Rep, Period>& rel_time);
```

10 *Requires:* If `mutex_type` is not a recursive mutex the calling thread does not own the mutex. The supplied `Mutex` type shall meet the `TimedLockable` requirements (30.21.0.1).

11 *Effects:* Constructs an object of type `unique_lock` and calls `m.try_lock_for(rel_time)`.

12 *Postconditions:* `pm == &m` and `owns == res`, where `res` is the value returned by the call to `m.try_lock_for(rel_time)`.

```
unique_lock(unique_lock&& u) noexcept;
```

13 *Postconditions:* `pm == u_p.pm` and `owns == u_p.owns` (where `u_p` is the state of `u` just prior to this construction), `u.pm == 0` and `u.owns == false`.

```
unique_lock& operator=(unique_lock&& u);
```

14 *Effects:* If `owns` calls `pm->unlock()`.

15 *Postconditions:* `pm == u_p.pm` and `owns == u_p.owns` (where `u_p` is the state of `u` just prior to this construction), `u.pm == 0` and `u.owns == false`.

16 [*Note:* With a recursive mutex it is possible for both `*this` and `u` to own the same mutex before the assignment. In this case, `*this` will own the mutex after the assignment and `u` will not. — *end note*]

17 *Throws:* Nothing.

```
~unique_lock();
```

18 *Effects:* If `owns` calls `pm->unlock()`.

30.146.0.0.1 `unique_lock` locking [thread.lock.unique.locking]

```
void lock();
```

1 *Effects:* `pm->lock()`

2 *Postcondition:* `owns == true`

3 *Throws:* Any exception thrown by `pm->lock()`. `system_error` if an exception is required (30.3.2). `system_error` with an error condition of `operation_not_permitted` if `pm` is 0. `system_error` with an error condition of `resource_deadlock_would_occur` if on entry `owns` is `true`.

```
bool try_lock();
```

4 *Requires:* The supplied `Mutex` shall meet the `Lockable` requirements (30.19.0.1).

5 *Effects:* `pm->try_lock()`

6 *Returns:* The value returned by the call to `try_lock()`.

7 *Postcondition:* `owns == res`, where `res` is the value returned by the call to `try_lock()`.

8 *Throws:* Any exception thrown by `pm->try_lock()`. `system_error` if an exception is required (30.3.2). `system_error` with an error condition of `operation_not_permitted` if `pm` is 0. `system_error` with an error condition of `resource_deadlock_would_occur` if on entry `owns` is `true`.

```
template <class Clock, class Duration>
  bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);
```

9 *Requires:* The supplied `Mutex` type shall meet the `TimedLockable` requirements (30.21.0.1).

10 *Effects:* `pm->try_lock_until(abs_time)`

11 *Returns:* The value returned by the call to `try_lock_until(abs_time)`.

12 *Postcondition:* `owns == res`, where `res` is the value returned by the call to `try_lock_until(abs_time)`.

13 *Throws:* Any exception thrown by `pm->try_lock_until()`. `system_error` if an exception is required (30.3.2). `system_error` with an error condition of `operation_not_permitted` if `pm` is 0. `system_error` with an error condition of `resource_deadlock_would_occur` if on entry `owns` is `true`.

```
template <class Rep, class Period>
  bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);
```

14 *Requires:* The supplied `Mutex` type shall meet the `TimedLockable` requirements (30.21.0.1).

15 *Effects:* `pm->try_lock_for(rel_time)`.

16 *Returns:* The value returned by the call to `try_lock_until(rel_time)`.

17 *Postcondition:* `owns == res`, where `res` is the value returned by the call to `try_lock_for(rel_time)`.

18 *Throws:* Any exception thrown by `pm->try_lock_for()`. `system_error` if an exception is required (30.3.2). `system_error` with an error condition of `operation_not_permitted` if `pm` is 0. `system_error` with an error condition of `resource_deadlock_would_occur` if on entry `owns` is `true`.

```
void unlock();
```

19 *Effects:* `pm->unlock()`

20 *Postcondition:* `owns == false`

21 *Throws:* `system_error` when an exception is required (30.3.2).

22 *Error conditions:*

(22.1) — `operation_not_permitted` — if on entry `owns` is false.

30.146.0.0.2 `unique_lock` modifiers [thread.lock.unique.mod]

```
void swap(unique_lock& u) noexcept;
```

1 *Effects:* Swaps the data members of `*this` and `u`.

```
mutex_type* release() noexcept;
```

2 *Returns:* The previous value of `pm`.

3 *Postconditions:* `pm == 0` and `owns == false`.

```
template <class Mutex>
  void swap(unique_lock<Mutex>& x, unique_lock<Mutex>& y) noexcept;
```

4 *Effects:* `x.swap(y)`

30.146.0.0.3 `unique_lock` observers [thread.lock.unique.obs]

```
bool owns_lock() const noexcept;
```

1 *Returns:* `owns`

```
explicit operator bool() const noexcept;
```

2 *Returns:* `owns`

```
mutex_type *mutex() const noexcept;
```

3 *Returns:* `pm`

30.146.0.1 Class template `shared_lock` [thread.lock.shared]

```
  namespace std {

  template <class Mutex>
  class shared_lock {
  public:
    typedef Mutex mutex_type;

    // Shared locking
    shared_lock() noexcept;
    explicit shared_lock(mutex_type& m);  // blocking
    shared_lock(mutex_type& m, defer_lock_t) noexcept;
    shared_lock(mutex_type& m, try_to_lock_t);
    shared_lock(mutex_type& m, adopt_lock_t);
    template <class Clock, class Duration>
      shared_lock(mutex_type& m,
                  const chrono::time_point<Clock, Duration>& abs_time);
    template <class Rep, class Period>
      shared_lock(mutex_type& m,
                  const chrono::duration<Rep, Period>& rel_time);
    ~shared_lock();

    shared_lock(shared_lock const&) = delete;
    shared_lock& operator=(shared_lock const&) = delete;

    shared_lock(shared_lock&& u) noexcept;
    shared_lock& operator=(shared_lock&& u) noexcept;
```

```
    void lock();  // blocking
    bool try_lock();
    template <class Rep, class Period>
      bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);
    template <class Clock, class Duration>
      bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);
    void unlock();

    // Setters
    void swap(shared_lock& u) noexcept;
    mutex_type* release() noexcept;

    // Getters
    bool owns_lock() const noexcept;
    explicit operator bool () const noexcept;
    mutex_type* mutex() const noexcept;

  private:
    mutex_type* pm; // exposition only
    bool owns;      // exposition only
  };

  template <class Mutex>
    void swap(shared_lock<Mutex>& x, shared_lock<Mutex>& y) noexcept;

  }  // std
```

1 An object of type `shared_lock` controls the shared ownership of a lockable object within a scope. Shared ownership of the lockable object may be acquired at construction or after construction, and may be transferred, after acquisition, to another `shared_lock` object. Objects of type `shared_lock` are not copyable but are movable. The behavior of a program is undefined if the contained pointer `pm` is not null and the lockable object pointed to by `pm` does not exist for the entire remaining lifetime (3.2563) of the `shared_lock` object. The supplied `Mutex` type shall meet the shared mutex requirements (30.135.0.2).

2 [*Note:* `shared_lock<Mutex>` meets the `TimedLockable` requirements (30.21.0.1). — *end note*]

30.146.0.1.1 `shared_lock` constructors, destructor, and assignment [thread.lock.shared.cons]

`shared_lock() noexcept;`

1 *Effects:* Constructs an object of type `shared_lock`.

2 *Postconditions:* `pm == nullptr` and `owns == false`.

`explicit shared_lock(mutex_type& m);`

3 *Requires:* The calling thread does not own the mutex for any ownership mode.

4 *Effects:* Constructs an object of type `shared_lock` and calls `m.lock_shared()`.

5 *Postconditions:* `pm == &m` and `owns == true`.

`shared_lock(mutex_type& m, defer_lock_t) noexcept;`

6 *Effects:* Constructs an object of type `shared_lock`.

7 *Postconditions:* `pm == &m` and `owns == false`.

`shared_lock(mutex_type& m, try_to_lock_t);`

8 *Requires:* The calling thread does not own the mutex for any ownership mode.

9 *Effects:* Constructs an object of type `shared_lock` and calls `m.try_lock_shared()`.

10 *Postconditions:* `pm == &m` and `owns == res` where `res` is the value returned by the call to `m.try_lock_shared()`.

```
shared_lock(mutex_type& m, adopt_lock_t);
```

11 *Requires:* The calling thread has shared ownership of the mutex.

12 *Effects:* Constructs an object of type `shared_lock`.

13 *Postconditions:* `pm == &m` and `owns == true`.

```
template <class Clock, class Duration>
  shared_lock(mutex_type& m,
              const chrono::time_point<Clock, Duration>& abs_time);
```

14 *Requires:* The calling thread does not own the mutex for any ownership mode.

15 *Effects:* Constructs an object of type `shared_lock` and calls `m.try_lock_shared_until(abs_time)`.

16 *Postconditions:* `pm == &m` and `owns == res` where `res` is the value returned by the call to `m.try_lock_shared_until(abs_time)`.

```
template <class Rep, class Period>
  shared_lock(mutex_type& m,
              const chrono::duration<Rep, Period>& rel_time);
```

17 *Requires:* The calling thread does not own the mutex for any ownership mode.

18 *Effects:* Constructs an object of type `shared_lock` and calls `m.try_lock_shared_for(rel_time)`.

19 *Postconditions:* `pm == &m` and `owns == res` where `res` is the value returned by the call to `m.try_lock_shared_for(rel_time)`.

```
~shared_lock();
```

20 *Effects:* If `owns` calls `pm->unlock_shared()`.

```
shared_lock(shared_lock&& sl) noexcept;
```

21 *Postconditions:* `pm == &sl_p.pm` and `owns == sl_p.owns` (where `sl_p` is the state of `sl` just prior to this construction), `sl.pm == nullptr` and `sl.owns == false`.

```
shared_lock& operator=(shared_lock&& sl) noexcept;
```

22 *Effects:* If `owns` calls `pm->unlock_shared()`.

23 *Postconditions:* `pm == &sl_p.pm` and `owns == sl_p.owns` (where `sl_p` is the state of `sl` just prior to this assignment), `sl.pm == nullptr` and `sl.owns == false`.

30.146.0.1.2 `shared_lock` locking [thread.lock.shared.locking]

```
void lock();
```

1 *Effects:* `pm->lock_shared()`.

2 *Postconditions:* `owns == true`.

3 *Throws:* Any exception thrown by `pm->lock_shared()`. `system_error` if an exception is required (30.3.2). `system_error` with an error condition of `operation_not_permitted` if `pm` is `nullptr`. `system_error` with an error condition of `resource_deadlock_would_occur` if on entry `owns` is `true`.

```
bool try_lock();
```

4 *Effects:* `pm->try_lock_shared()`.

5 *Returns:* The value returned by the call to `pm->try_lock_shared()`.

6 *Postconditions:* `owns == res`, where `res` is the value returned by the call to `pm->try_lock_shared()`.

7 *Throws:* Any exception thrown by `pm->try_lock_shared()`. `system_error` if an exception is required (30.3.2). `system_error` with an error condition of `operation_not_permitted` if `pm` is `nullptr`. `system_error` with an error condition of `resource_deadlock_would_occur` if on entry `owns` is `true`.

```
template <class Clock, class Duration>
  bool
  try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);
```

8 *Effects:* `pm->try_lock_shared_until(abs_time)`.

9 *Returns:* The value returned by the call to `pm->try_lock_shared_until(abs_time)`.

10 *Postconditions:* `owns == res`, where `res` is the value returned by the call to `pm->try_lock_shared_until(abs_time)`.

11 *Throws:* Any exception thrown by `pm->try_lock_shared_until(abs_time)`. `system_error` if an exception is required (30.3.2). `system_error` with an error condition of `operation_not_permitted` if `pm` is `nullptr`. `system_error` with an error condition of `resource_deadlock_would_occur` if on entry `owns` is `true`.

```
template <class Rep, class Period>
  bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);
```

12 *Effects:* `pm->try_lock_shared_for(rel_time)`.

13 *Returns:* The value returned by the call to `pm->try_lock_shared_for(rel_time)`.

14 *Postconditions:* `owns == res`, where `res` is the value returned by the call to `pm->try_lock_shared_for(rel_time)`.

15 *Throws:* Any exception thrown by `pm->try_lock_shared_for(rel_time)`. `system_error` if an exception is required (30.3.2). `system_error` with an error condition of `operation_not_permitted` if `pm` is `nullptr`. `system_error` with an error condition of `resource_deadlock_would_occur` if on entry `owns` is `true`.

```
void unlock();
```

16 *Effects:* `pm->unlock_shared()`.

17 *Postconditions:* `owns == false`.

18 *Throws:* `system_error` when an exception is required (30.3.2).

19 *Error conditions:*

(19.1) — `operation_not_permitted` — if on entry `owns` is `false`.

30.146.0.1.3 `shared_lock` modifiers [thread.lock.shared.mod]

```
void swap(shared_lock& sl) noexcept;
```

1 *Effects:* Swaps the data members of `*this` and `sl`.

```
mutex_type* release() noexcept;
```

2 *Returns:* The previous value of `pm`.

3 *Postconditions:* `pm == nullptr` and `owns == false`.

```
template <class Mutex>
  void swap(shared_lock<Mutex>& x, shared_lock<Mutex>& y) noexcept;
```

4 *Effects:* `x.swap(y)`.

30.146.0.1.4 `shared_lock` observers [thread.lock.shared.obs]

```
bool owns_lock() const noexcept;
```

1 *Returns:* `owns`.

```
explicit operator bool () const noexcept;
```

2 *Returns:* `owns`.

```
mutex_type* mutex() const noexcept;
```

3 *Returns:* `pm`.

30.146.1 Generic locking algorithms [thread.lock.algorithm]

```
template <class L1, class L2, class... L3> int try_lock(L1&, L2&, L3&...);
```

1

Requires: Each template parameter type shall meet the `Lockable` requirements. [*Note:* The `unique_lock` class template meets these requirements when suitably instantiated. — *end note*]

1 *Effects:* Calls `try_lock()` for each argument in order beginning with the first until all arguments have been processed or a call to `try_lock()` fails, either by returning `false` or by throwing an exception. If a call to `try_lock()` fails, `unlock()` shall be called for all prior arguments and there shall be no further calls to `try_lock()`.

2 *Returns:* `-1` if all calls to `try_lock()` returned `true`, otherwise a 0-based index value that indicates the argument for which `try_lock()` returned `false`.

```
template <class L1, class L2, class... L3> void lock(L1&, L2&, L3&...);
```

3 *Requires:* Each template parameter type shall meet the `Lockable` requirements, [*Note:* The `unique_lock` class template meets these requirements when suitably instantiated. — *end note*]

4

Effects: All arguments are locked via a sequence of calls to `lock()`, `try_lock()`, or `unlock()` on each argument. The sequence of calls shall not result in deadlock, but is otherwise unspecified. [*Note:* A deadlock avoidance algorithm such as try-and-back-off must be used, but the specific algorithm is not specified to avoid over-constraining implementations. — *end note*] If a call to `lock()` or `try_lock()` throws an exception, `unlock()` shall be called for any argument that had been locked by a call to `lock()` or `try_lock()`.

30.150.1 Call once [thread.once]

The class `once_flag` is an opaque data structure that `call_once` uses to initialize data without causing a data race or deadlock.

30.150.1.1 Struct `once_flag` [thread.once.onceflag]

```
constexpr once_flag() noexcept;
```

1 *Effects:* Constructs an object of type `once_flag`.

2 *Synchronization:* The construction of a `once_flag` object is not synchronized.

3 *Postcondition:* The object's internal state is set to indicate to an invocation of `call_once` with the object as its initial argument that no function has been called.

30.150.1.2 Function `call_once` [thread.once.callonce]

```
template<class Callable, class ...Args>
  void call_once(once_flag& flag, Callable&& func, Args&&... args);
```

1 *Requires:* `Callable` and each `Ti` in `Args` shall satisfy the `MoveConstructible` requirements. *`INVOKE`*`(`*`DECAY_COPY`*`( std::forward<Callable>(func)), `*`DECAY_COPY`*`(std::forward<Args>(args))...)` (20.295.1) shall be a valid expression.

2 *Effects:* An execution of `call_once` that does not call its `func` is a *passive* execution. An execution of `call_once` that calls its `func` is an *active* execution. An active execution shall call *`INVOKE`*`(`*`DECAY_COPY`*`( std::forward<Callable>(func)), `*`DECAY_COPY`*`(std::forward<Args>(args))...)`. If such a call to `func` throws an exception the execution is *exceptional*, otherwise it is *returning*. An exceptional execution shall propagate the exception to the caller of `call_once`. Among all executions of `call_once` for any given `once_flag`: at most one shall be a returning execution; if there is a returning execution, it shall be the last active execution; and there are passive executions only if there is a returning execution. [*Note:* passive executions allow other threads to reliably observe the results produced by the earlier returning execution. — *end note*]

1 *Synchronization:* For any given `once_flag`: all active executions occur in a total order; completion of an active execution synchronizes with (1.859) the start of the next one in this total order; and the returning execution synchronizes with the return from all passive executions.

1 *Throws:* `system_error` when an exception is required (30.3.2), or any exception thrown by `func`.

2 [*Example:*

```
// global flag, regular function
void init();
std::once_flag flag;

void f() {
  std::call_once(flag, init);
}

// function static flag, function object
struct initializer {
  void operator()();
};

void g() {
  static std::once_flag flag2;
  std::call_once(flag2, initializer());
}

// object flag, member function
class information {
  std::once_flag verified;
  void verifier();
public:
  void verify() { std::call_once(verified, &information::verifier, *this); }
};
```

— *end example*]

30.157 Condition variables [thread.condition]

1

Condition variables provide synchronization primitives used to block a thread until notified by some other thread that some condition is met or until a system time is reached. Class `condition_variable` provides a condition variable that can only wait on an object of type `unique_lock<mutex>`, allowing maximum efficiency on some platforms. Class `condition_variable_any` provides a general condition variable that can wait on objects of user-supplied lock types.

1 Condition variables permit concurrent invocation of the `wait`, `wait_for`, `wait_until`, `notify_one` and `notify_all` member functions.

2

The execution of `notify_one` and `notify_all` shall be atomic. The execution of `wait`, `wait_for`, and `wait_until` shall be performed in three atomic parts:

1. the release of the mutex and entry into the waiting state;
2. the unblocking of the wait; and
3. the reacquisition of the lock.

1 The implementation shall behave as if all executions of `notify_one`, `notify_all`, and each part of the `wait`, `wait_for`, and `wait_until` executions are executed in a single unspecified total order consistent with the "happens before" order.

2 Condition variable construction and destruction need not be synchronized.

Header `condition_variable` synopsis

```
namespace std {
  class condition_variable;
  class condition_variable_any;

  void notify_all_at_thread_exit(condition_variable& cond, unique_lock<mutex> lk);

  enum class cv_status { no_timeout, timeout };
}
```

```
void notify_all_at_thread_exit(condition_variable& cond, unique_lock<mutex> lk);
```

3 *Requires:* `lk` is locked by the calling thread and either

(3.1) — no other thread is waiting on `cond`, or

(3.2) — `lk.mutex()` returns the same value for each of the lock arguments supplied by all concurrently waiting (via `wait`, `wait_for`, or `wait_until`) threads.

4 *Effects:* transfers ownership of the lock associated with `lk` into internal storage and schedules `cond` to be notified when the current thread exits, after all objects of thread storage duration associated with the current thread have been destroyed. This notification shall be as if

```
lk.unlock();
cond.notify_all();
```

5 *Synchronization:* The implied `lk.unlock()` call is sequenced after the destruction of all objects with thread storage duration associated with the current thread.

6 *Note:* The supplied lock will be held until the thread exits, and care must be taken to ensure that this does not cause deadlock due to lock ordering issues. After calling `notify_all_at_thread_exit`

it is recommended that the thread should be exited as soon as possible, and that no blocking or time-consuming tasks are run on that thread.

7

Note: It is the user's responsibility to ensure that waiting threads do not erroneously assume that the thread has finished if they experience spurious wakeups. This typically requires that the condition being waited for is satisfied while holding the lock on `lk`, and that this lock is not released and reacquired prior to calling `notify_all_at_thread_exit`.

30.161.1 Class `condition_variable` [thread.condition.condvar]

```
namespace std {
  class condition_variable {
  public:

    condition_variable();
    ~condition_variable();

    condition_variable(const condition_variable&) = delete;
    condition_variable& operator=(const condition_variable&) = delete;

    void notify_one() noexcept;
    void notify_all() noexcept;
    void wait(unique_lock<mutex>& lock);
    template <class Predicate>
      void wait(unique_lock<mutex>& lock, Predicate pred);
    template <class Clock, class Duration>
      cv_status wait_until(unique_lock<mutex>& lock,
                           const chrono::time_point<Clock, Duration>& abs_time);
    template <class Clock, class Duration, class Predicate>
      bool wait_until(unique_lock<mutex>& lock,
                      const chrono::time_point<Clock, Duration>& abs_time,
                      Predicate pred);

    template <class Rep, class Period>
      cv_status wait_for(unique_lock<mutex>& lock,
                         const chrono::duration<Rep, Period>& rel_time);
    template <class Rep, class Period, class Predicate>
      bool wait_for(unique_lock<mutex>& lock,
                    const chrono::duration<Rep, Period>& rel_time,
                    Predicate pred);

    typedef implementation-defined native_handle_type; // See 30.6.1
    native_handle_type native_handle();                // See 30.6.1
  };
}
```

1 The class `condition_variable` shall be a standard-layout class (Clause 9).

`condition_variable();`

2 *Effects:* Constructs an object of type `condition_variable`.

3 *Throws:* `system_error` when an exception is required (30.3.2).

4 *Error conditions:*

(4.1) — `resource_unavailable_try_again` — if some non-memory resource limitation prevents initialization.

```
~condition_variable();
```

5

Requires: There shall be no thread blocked on `*this`. [*Note:* That is, all threads shall have been notified; they may subsequently block on the lock specified in the wait. This relaxes the usual rules, which would have required all wait calls to happen before destruction. Only the notification to unblock the wait must happen before destruction. The user must take care to ensure that no threads wait on `*this` once the destructor has been started, especially when the waiting threads are calling the wait functions in a loop or using the overloads of `wait`, `wait_for`, or `wait_until` that take a predicate. — *end note*]

1 *Effects:* Destroys the object.

```
void notify_one() noexcept;
```

2 *Effects:* If any threads are blocked waiting for `*this`, unblocks one of those threads.

```
void notify_all() noexcept;
```

1 *Effects:* Unblocks all threads that are blocked waiting for `*this`.

```
void wait(unique_lock<mutex>& lock);
```

1

Requires: `lock.owns_lock()` is `true` and `lock.mutex()` is locked by the calling thread, and either

— no other thread is waiting on this `condition_variable` object or

— `lock.mutex()` returns the same value for each of the `lock` arguments supplied by all concurrently waiting (via `wait`, `wait_for`, or `wait_until`) threads.

1 *Effects:*

— Atomically calls `lock.unlock()` and blocks on `*this`.

— When unblocked, calls `lock.lock()` (possibly blocking on the lock), then returns.

— The function will unblock when signaled by a call to `notify_one()` or a call to `notify_all()`, or spuriously.

1 *Remarks:* If the function fails to meet the postcondition, `std::terminate()` shall be called (15.308.1). [*Note:* This can happen if the re-locking of the mutex throws an exception. — *end note*]

2 *Postcondition:* `lock.owns_lock()` is `true` and `lock.mutex()` is locked by the calling thread.

3 *Throws:* Nothing.

```
template <class Predicate>
  void wait(unique_lock<mutex>& lock, Predicate pred);
```

4 *Requires:* `lock.owns_lock()` is `true` and `lock.mutex()` is locked by the calling thread, and either

(4.1) — no other thread is waiting on this `condition_variable` object or

(4.2) — `lock.mutex()` returns the same value for each of the `lock` arguments supplied by all concurrently waiting (via `wait`, `wait_for`, or `wait_until`) threads.

5 *Effects:* Equivalent to:

```
while (!pred())
  wait(lock);
```

6 *Remarks:* If the function fails to meet the postcondition, `std::terminate()` shall be called (15.308.1). [*Note:* This can happen if the re-locking of the mutex throws an exception. — *end note*]

7 *Postcondition:* `lock.owns_lock()` is `true` and `lock.mutex()` is locked by the calling thread.

8 *Throws:* Timeout-related exceptions (30.8.1) or any exception thrown by `pred`.

```
template <class Clock, class Duration>
  cv_status wait_until(unique_lock<mutex>& lock,
                       const chrono::time_point<Clock, Duration>& abs_time);
```

9 *Requires:* `lock.owns_lock()` is `true` and `lock.mutex()` is locked by the calling thread, and either

(9.1) — no other thread is waiting on this `condition_variable` object or

(9.2) — `lock.mutex()` returns the same value for each of the `lock` arguments supplied by all concurrently waiting (via `wait`, `wait_for`, or `wait_until`) threads.

10

Effects:

— Atomically calls `lock.unlock()` and blocks on `*this`.

— When unblocked, calls `lock.lock()` (possibly blocking on the lock), then returns.

— The function will unblock when signaled by a call to `notify_one()`, a call to `notify_all()`, expiration of the absolute timeout (30.8.1) specified by `abs_time`, or spuriously.

— If the function exits via an exception, `lock.lock()` shall be called prior to exiting the function.

1 *Remarks:* If the function fails to meet the postcondition, `std::terminate()` shall be called (15.308.1). [*Note:* This can happen if the re-locking of the mutex throws an exception. — *end note*]

2 *Postcondition:* `lock.owns_lock()` is `true` and `lock.mutex()` is locked by the calling thread.

3

Returns: `cv_status::timeout` if the absolute timeout (30.8.1) specified by `abs_time` expired, otherwise `cv_status::no_timeout`.

1 *Throws:* Timeout-related exceptions (30.8.1).

```
template <class Rep, class Period>
  cv_status wait_for(unique_lock<mutex>& lock,
                     const chrono::duration<Rep, Period>& rel_time);
```

2 *Requires:* `lock.owns_lock()` is `true` and `lock.mutex()` is locked by the calling thread, and either

(2.1) — no other thread is waiting on this `condition_variable` object or

(2.2) — `lock.mutex()` returns the same value for each of the `lock` arguments supplied by all concurrently waiting (via `wait`, `wait_for`, or `wait_until`) threads.

3 *Effects:* Equivalent to:

```
return wait_until(lock, chrono::steady_clock::now() + rel_time);
```

4 *Returns:* `cv_status::timeout` if the relative timeout (30.8.1) specified by `rel_time` expired, otherwise `cv_status::no_timeout`.

5 *Remarks:* If the function fails to meet the postcondition, `std::terminate()` shall be called (15.308.1). [*Note:* This can happen if the re-locking of the mutex throws an exception. — *end note*]

6 *Postcondition:* `lock.owns_lock()` is `true` and `lock.mutex()` is locked by the calling thread.

7 *Throws:* Timeout-related exceptions (30.8.1).

```
template <class Clock, class Duration, class Predicate>
  bool wait_until(unique_lock<mutex>& lock,
                  const chrono::time_point<Clock, Duration>& abs_time,
                  Predicate pred);
```

8 *Requires:* `lock.owns_lock()` is `true` and `lock.mutex()` is locked by the calling thread, and either

(8.1) — no other thread is waiting on this `condition_variable` object or

(8.2) — `lock.mutex()` returns the same value for each of the `lock` arguments supplied by all concurrently waiting (via `wait`, `wait_for`, or `wait_until`) threads.

9 *Effects:* Equivalent to:

```
while (!pred())
  if (wait_until(lock, abs_time) == cv_status::timeout)
    return pred();
return true;
```

10 *Remarks:* If the function fails to meet the postcondition, `std::terminate()` shall be called (15.308.1). [*Note:* This can happen if the re-locking of the mutex throws an exception. — *end note*]

11 *Postcondition:* `lock.owns_lock()` is `true` and `lock.mutex()` is locked by the calling thread.

12 [*Note:* The returned value indicates whether the predicate evaluated to `true` regardless of whether the timeout was triggered. — *end note*]

13 *Throws:* Timeout-related exceptions (30.8.1) or any exception thrown by `pred`.

```
template <class Rep, class Period, class Predicate>
  bool wait_for(unique_lock<mutex>& lock,
                const chrono::duration<Rep, Period>& rel_time,
                Predicate pred);
```

14 *Requires:* `lock.owns_lock()` is `true` and `lock.mutex()` is locked by the calling thread, and either

(14.1) — no other thread is waiting on this `condition_variable` object or

(14.2) — `lock.mutex()` returns the same value for each of the `lock` arguments supplied by all concurrently waiting (via `wait`, `wait_for`, or `wait_until`) threads.

15 *Effects:* Equivalent to:

```
return wait_until(lock, chrono::steady_clock::now() + rel_time, std::move(pred));
```

16 [*Note:* There is no blocking if `pred()` is initially `true`, even if the timeout has already expired. — *end note*]

17 *Remarks:* If the function fails to meet the postcondition, `std::terminate()` shall be called (15.308.1). [*Note:* This can happen if the re-locking of the mutex throws an exception. — *end note*]

18 *Postcondition:* `lock.owns_lock()` is `true` and `lock.mutex()` is locked by the calling thread.

19 [*Note:* The returned value indicates whether the predicate evaluates to `true` regardless of whether the timeout was triggered. — *end note*]

20 *Throws:* Timeout-related exceptions (30.8.1) or any exception thrown by `pred`.

30.177.1 Class `condition_variable_any` [thread.condition.condvarany]

1

A `Lock` type shall meet the `BasicLockable` requirements (30.17.0.1). [*Note:* All of the standard mutex types meet this requirement. If a `Lock` type other than one of the standard mutex types or a `unique_lock` wrapper for a standard mutex type is used with `condition_variable_any`, the user must ensure that any necessary synchronization is in place with respect to the predicate associated with the `condition_variable_any` instance. — *end note*]

```
namespace std {
  class condition_variable_any {
  public:
    condition_variable_any();
    ~condition_variable_any();

    condition_variable_any(const condition_variable_any&) = delete;
    condition_variable_any& operator=(const condition_variable_any&) = delete;

    void notify_one() noexcept;
    void notify_all() noexcept;
    template <class Lock>
      void wait(Lock& lock);
    template <class Lock, class Predicate>
      void wait(Lock& lock, Predicate pred);

    template <class Lock, class Clock, class Duration>
      cv_status wait_until(Lock& lock, const chrono::time_point<Clock, Duration>& abs_time);
    template <class Lock, class Clock, class Duration, class Predicate>
      bool wait_until(Lock& lock, const chrono::time_point<Clock, Duration>& abs_time,
        Predicate pred);
    template <class Lock, class Rep, class Period>
      cv_status wait_for(Lock& lock, const chrono::duration<Rep, Period>& rel_time);
    template <class Lock, class Rep, class Period, class Predicate>
      bool wait_for(Lock& lock, const chrono::duration<Rep, Period>& rel_time,
        Predicate pred);
  };
}
```

`condition_variable_any();`

1 *Effects:* Constructs an object of type `condition_variable_any`.

2 *Throws:* `bad_alloc` or `system_error` when an exception is required (30.3.2).

3 *Error conditions:*

(3.1) — `resource_unavailable_try_again` — if some non-memory resource limitation prevents initialization.

(3.2) — `operation_not_permitted` — if the thread does not have the privilege to perform the operation.

`~condition_variable_any();`

4 *Requires:* There shall be no thread blocked on `*this`. [*Note:* That is, all threads shall have been notified; they may subsequently block on the lock specified in the wait. This relaxes the usual rules, which would have required all wait calls to happen before destruction. Only the notification to unblock the wait must happen before destruction. The user must take care to ensure that no threads wait on

`*this` once the destructor has been started, especially when the waiting threads are calling the wait functions in a loop or using the overloads of `wait`, `wait_for`, or `wait_until` that take a predicate. — *end note*]

5 *Effects:* Destroys the object.

```
void notify_one() noexcept;
```

6 *Effects:* If any threads are blocked waiting for `*this`, unblocks one of those threads.

```
void notify_all() noexcept;
```

7 *Effects:* Unblocks all threads that are blocked waiting for `*this`.

```
template <class Lock>
  void wait(Lock& lock);
```

8 *Note:* if any of the `wait` functions exits via an exception, it is unspecified whether the `Lock` is held. One can use a `Lock` type that allows to query that, such as the `unique_lock` wrapper.

9 *Effects:*

(9.1) — Atomically calls `lock.unlock()` and blocks on `*this`.

(9.2) — When unblocked, calls `lock.lock()` (possibly blocking on the lock) and returns.

(9.3) — The function will unblock when signaled by a call to `notify_one()`, a call to `notify_all()`, or spuriously.

10 *Remarks:* If the function fails to meet the postcondition, `std::terminate()` shall be called (15.308.1). [*Note:* This can happen if the re-locking of the mutex throws an exception. — *end note*]

11 *Postcondition:* `lock` is locked by the calling thread.

12 *Throws:* Nothing.

```
template <class Lock, class Predicate>
  void wait(Lock& lock, Predicate pred);
```

13 *Effects:* Equivalent to:

```
while (!pred())
  wait(lock);
```

```
template <class Lock, class Clock, class Duration>
  cv_status wait_until(Lock& lock, const chrono::time_point<Clock, Duration>& abs_time);
```

14 *Effects:*

(14.1) — Atomically calls `lock.unlock()` and blocks on `*this`.

(14.2) — When unblocked, calls `lock.lock()` (possibly blocking on the lock) and returns.

(14.3) — The function will unblock when signaled by a call to `notify_one()`, a call to `notify_all()`, expiration of the absolute timeout (30.8.1) specified by `abs_time`, or spuriously.

(14.4) — If the function exits via an exception, `lock.lock()` shall be called prior to exiting the function.

15 *Remarks:* If the function fails to meet the postcondition, `std::terminate()` shall be called (15.308.1). [*Note:* This can happen if the re-locking of the mutex throws an exception. — *end note*]

16 *Postcondition:* `lock` is locked by the calling thread.

17 *Returns:* `cv_status::timeout` if the absolute timeout (30.8.1) specified by `abs_time` expired, otherwise `cv_status::no_timeout`.

18 *Throws:* Timeout-related exceptions (30.8.1).

```
template <class Lock, class Rep, class Period>
  cv_status wait_for(Lock& lock, const chrono::duration<Rep, Period>& rel_time);
```

19 *Effects:* Equivalent to:

```
return wait_until(lock, chrono::steady_clock::now() + rel_time);
```

20 *Returns:* `cv_status::timeout` if the relative timeout (30.8.1) specified by `rel_time` expired, otherwise `cv_status::no_timeout`.

21 *Remarks:* If the function fails to meet the postcondition, `std::terminate()` shall be called (15.308.1). [*Note:* This can happen if the re-locking of the mutex throws an exception. — *end note*]

22 *Postcondition:* `lock` is locked by the calling thread.

23 *Throws:* Timeout-related exceptions (30.8.1).

```
template <class Lock, class Clock, class Duration, class Predicate>
  bool wait_until(Lock& lock, const chrono::time_point<Clock, Duration>& abs_time, Predicate pred);
```

24 *Effects:* Equivalent to:

```
while (!pred())
  if (wait_until(lock, abs_time) == cv_status::timeout)
    return pred();
return true;
```

25 [*Note:* There is no blocking if `pred()` is initially `true`, or if the timeout has already expired. — *end note*]

26 [*Note:* The returned value indicates whether the predicate evaluates to `true` regardless of whether the timeout was triggered. — *end note*]

```
template <class Lock, class Rep, class Period, class Predicate>
  bool wait_for(Lock& lock, const chrono::duration<Rep, Period>& rel_time, Predicate pred);
```

27 *Effects:* Equivalent to:

```
return wait_until(lock, chrono::steady_clock::now() + rel_time, std::move(pred));
```

30.179 Futures [futures]

30.179.1 Overview [futures.overview]

1 30.179 describes components that a C++ program can use to retrieve in one thread the result (value or exception) from a function that has run in the same thread or another thread. [*Note:* These components are not restricted to multi-threaded programs but can be useful in single-threaded programs as well. — *end note*]

Header `<future>` synopsis

```
namespace std {
  enum class future_errc {
    broken_promise = implementation-defined,
    future_already_retrieved = implementation-defined,
    promise_already_satisfied = implementation-defined,
    no_state = implementation-defined
  };

  enum class launch : unspecified {
    async = unspecified,
```

```
    deferred = unspecified,
    implementation-defined
  };

  enum class future_status {
    ready,
    timeout,
    deferred
  };

  template <> struct is_error_code_enum<future_errc> : public true_type { };
  error_code make_error_code(future_errc e) noexcept;
  error_condition make_error_condition(future_errc e) noexcept;

  const error_category& future_category() noexcept;

  class future_error;

  template <class R> class promise;
  template <class R> class promise<R&>;
  template <> class promise<void>;

  template <class R>
    void swap(promise<R>& x, promise<R>& y) noexcept;

  template <class R, class Alloc>
    struct uses_allocator<promise<R>, Alloc>;

  template <class R> class future;
  template <class R> class future<R&>;
  template <> class future<void>;

  template <class R> class shared_future;
  template <class R> class shared_future<R&>;
  template <> class shared_future<void>;

  template <class> class packaged_task;   // undefined
  template <class R, class... ArgTypes>
    class packaged_task<R(ArgTypes...)>;

  template <class R, class... ArgTypes>
    void swap(packaged_task<R(ArgTypes...)>&, packaged_task<R(ArgTypes...)>&) noexcept;

  template <class R, class Alloc>
    struct uses_allocator<packaged_task<R>, Alloc>;

  template <class F, class... Args>
    future<result_of_t<decay_t<F>(decay_t<Args>...)>>
    async(F&& f, Args&&... args);
  template <class F, class... Args>
    future<result_of_t<decay_t<F>(decay_t<Args>...)>>
    async(launch policy, F&& f, Args&&... args);
}
```

2 The `enum` type `launch` is a bitmask type (17.20.0.0.1) with `launch::async` and `launch::deferred` denoting

individual bits. [*Note:* Implementations can provide bitmasks to specify restrictions on task interaction by functions launched by `async()` applicable to a corresponding subset of available launch policies. Implementations can extend the behavior of the first overload of `async()` by adding their extensions to the launch policy under the "as if" rule. — *end note*]

3 The enum values of `future_errc` are distinct and not zero.

30.179.2 Error handling [futures.errors]

```
const error_category& future_category() noexcept;
```

1 *Returns:* A reference to an object of a type derived from class `error_category`.

2 The object's `default_error_condition` and equivalent virtual functions shall behave as specified for the class `error_category`. The object's `name` virtual function shall return a pointer to the string `"future"`.

```
error_code make_error_code(future_errc e) noexcept;
```

3 *Returns:* `error_code(static_cast<int>(e), future_category())`.

```
error_condition make_error_condition(future_errc e) noexcept;
```

4 *Returns:* `error_condition(static_cast<int>(e), future_category())`.

30.179.3 Class `future_error` [futures.future_error]

```
namespace std {
  class future_error : public logic_error {
  public:
    future_error(error_code ec);  // exposition only

    const error_code& code() const noexcept;
    const char*       what() const noexcept;
  };
}
```

```
const error_code& code() const noexcept;
```

1 *Returns:* The value of `ec` that was passed to the object's constructor.

```
const char* what() const noexcept;
```

2 *Returns:* An NTBS incorporating `code().message()`.

30.179.4 Shared state [futures.state]

1

Many of the classes introduced in this sub-clause use some state to communicate results. This *shared state* consists of some state information and some (possibly not yet evaluated) *result*, which can be a (possibly void) value or an exception. [*Note:* Futures, promises, and tasks defined in this clause reference such shared state. — *end note*]

1

[*Note:* The result can be any kind of object including a function to compute that result, as used by `async` when `policy` is `launch::deferred`. — *end note*]

1 An *asynchronous return object* is an object that reads results from a shared state. A *waiting function* of an asynchronous return object is one that potentially blocks to wait for the shared state to be made ready. If a

waiting function can return before the state is made ready because of a timeout (30.16.1), then it is a *timed waiting function*, otherwise it is a *non-timed waiting function*.

2 An *asynchronous provider* is an object that provides a result to a shared state. The result of a shared state is set by respective functions on the asynchronous provider. [*Note:* Such as promises or tasks. — *end note*] The means of setting the result of a shared state is specified in the description of those classes and functions that create such a state object.

3

When an asynchronous return object or an asynchronous provider is said to release its shared state, it means:

— if the return object or provider holds the last reference to its shared state, the shared state is destroyed; and

— the return object or provider gives up its reference to its shared state; and

— these actions will not block for the shared state to become ready, except that it may block if all of the following are true: the shared state was created by a call to `std::async`, the shared state is not yet ready, and this was the last reference to the shared state.

1 When an asynchronous provider is said to make its shared state ready, it means:

(1.1) — first, the provider marks its shared state as ready; and

(1.2) — second, the provider unblocks any execution agents waiting for its shared state to become ready.

2 When an asynchronous provider is said to abandon its shared state, it means:

(2.1) — first, if that state is not ready, the provider

(2.1.1) — stores an exception object of type `future_error` with an error condition of `broken_promise` within its shared state; and then

(2.1.2) — makes its shared state ready;

(2.2) — second, the provider releases its shared state.

3 A shared state is *ready* only if it holds a value or an exception ready for retrieval. Waiting for a shared state to become ready may invoke code to compute the result on the waiting thread if so specified in the description of the class or function that creates the state object.

4

Calls to functions that successfully set the stored result of a shared state synchronize with (1.859) calls to functions successfully detecting the ready state resulting from that setting. The storage of the result (whether normal or exceptional) into the shared state synchronizes with (1.859) the successful return from a call to a waiting function on the shared state.

1 Some functions (e.g., `promise::set_value_at_thread_exit`) delay making the shared state ready until the calling thread exits. The destruction of each of that thread's objects with thread storage duration (3.2291.1) is sequenced before making that shared state ready.

2 Access to the result of the same shared state may conflict (1.859). [*Note:* this explicitly specifies that the result of the shared state is visible in the objects that reference this state in the sense of data race avoidance (17.607.0.1). For example, concurrent accesses through references returned by `shared_future::get()` (30.203.1) must either use read-only operations or provide additional synchronization. — *end note*]

30.191.1 Class template promise [futures.promise]

```
namespace std {
  template <class R>
  class promise {
  public:
    promise();
    template <class Allocator>
      promise(allocator_arg_t, const Allocator& a);
    promise(promise&& rhs) noexcept;
    promise(const promise& rhs) = delete;
    ~promise();

    // assignment
    promise& operator=(promise&& rhs) noexcept;
    promise& operator=(const promise& rhs) = delete;
    void swap(promise& other) noexcept;

    // retrieving the result
    future<R> get_future();

    // setting the result
    void set_value(see below);
    void set_exception(exception_ptr p);

    // setting the result with deferred notification
    void set_value_at_thread_exit(const R& r);
    void set_value_at_thread_exit(see below);
    void set_exception_at_thread_exit(exception_ptr p);
  };
  template <class R>
    void swap(promise<R>& x, promise<R>& y) noexcept;
  template <class R, class Alloc>
    struct uses_allocator<promise<R>, Alloc>;
}
```

1 The implementation shall provide the template `promise` and two specializations, `promise<R&>` and `promise<void>`. These differ only in the argument type of the member function `set_value`, as set out in its description, below.

2 The `set_value`, `set_exception`, `set_value_at_thread_exit`, and `set_exception_at_thread_exit` member functions behave as though they acquire a single mutex associated with the promise object while updating the promise object.

```
template <class R, class Alloc>
  struct uses_allocator<promise<R>, Alloc>
    : true_type { };
```

1 *Requires:* `Alloc` shall be an Allocator (17.165.0.1).

```
promise();
template <class Allocator>
  promise(allocator_arg_t, const Allocator& a);
```

2

Effects: constructs a `promise` object and a shared state. The second constructor uses the allocator `a` to allocate memory for the shared state.

```
promise(promise&& rhs) noexcept;
```

1 *Effects:* constructs a new `promise` object and transfers ownership of the shared state of `rhs` (if any) to the newly-constructed object.

2 *Postcondition:* `rhs` has no shared state.

```
~promise();
```

3 *Effects:* Abandons any shared state (30.179.4).

```
promise& operator=(promise&& rhs) noexcept;
```

4 *Effects:* Abandons any shared state (30.179.4) and then as if `promise(std::move(rhs)).swap(*this)`.

5 *Returns:* `*this`.

```
void swap(promise& other) noexcept;
```

6 *Effects:* Exchanges the shared state of `*this` and `other`.

7 *Postcondition:* `*this` has the shared state (if any) that `other` had prior to the call to `swap`. `other` has the shared state (if any) that `*this` had prior to the call to `swap`.

```
future<R> get_future();
```

8 *Returns:* A `future<R>` object with the same shared state as `*this`.

9 *Throws:* `future_error` if `*this` has no shared state or if `get_future` has already been called on a `promise` with the same shared state as `*this`.

10 *Error conditions:*

(10.1) — `future_already_retrieved` if `get_future` has already been called on a `promise` with the same shared state as `*this`.

(10.2) — `no_state` if `*this` has no shared state.

```
void promise::set_value(const R& r);
void promise::set_value(R&& r);
void promise<R&>::set_value(R& r);
void promise<void>::set_value();
```

11 *Effects:* atomically stores the value `r` in the shared state and makes that state ready (30.179.4).

12 *Throws:*

(12.1) — `future_error` if its shared state already has a stored value or exception, or

(12.2) — for the first version, any exception thrown by the constructor selected to copy an object of `R`, or

(12.3) — for the second version, any exception thrown by the constructor selected to move an object of `R`.

13 *Error conditions:*

(13.1) — `promise_already_satisfied` if its shared state already has a stored value or exception.

(13.2) — `no_state` if `*this` has no shared state.

```
void set_exception(exception_ptr p);
```

14 *Effects:* atomically stores the exception pointer `p` in the shared state and makes that state ready (30.179.4).

15 *Throws:* `future_error` if its shared state already has a stored value or exception.

16 *Error conditions:*

(16.1) — `promise_already_satisfied` if its shared state already has a stored value or exception.

(16.2) — `no_state` if `*this` has no shared state.

```
void promise::set_value_at_thread_exit(const R& r);
void promise::set_value_at_thread_exit(R&& r);
void promise<R&>::set_value_at_thread_exit(R& r);
void promise<void>::set_value_at_thread_exit();
```

17 *Effects:* Stores the value `r` in the shared state without making that state ready immediately. Schedules that state to be made ready when the current thread exits, after all objects of thread storage duration associated with the current thread have been destroyed.

18 *Throws:*

(18.1) — `future_error` if its shared state already has a stored value or exception, or

(18.2) — for the first version, any exception thrown by the constructor selected to copy an object of `R`, or

(18.3) — for the second version, any exception thrown by the constructor selected to move an object of `R`.

19 *Error conditions:*

(19.1) — `promise_already_satisfied` if its shared state already has a stored value or exception.

(19.2) — `no_state` if `*this` has no shared state.

```
void set_exception_at_thread_exit(exception_ptr p);
```

20 *Effects:* Stores the exception pointer `p` in the shared state without making that state ready immediately. Schedules that state to be made ready when the current thread exits, after all objects of thread storage duration associated with the current thread have been destroyed.

21 *Throws:* `future_error` if an error condition occurs.

22 *Error conditions:*

(22.1) — `promise_already_satisfied` if its shared state already has a stored value or exception.

(22.2) — `no_state` if `*this` has no shared state.

```
template <class R>
  void swap(promise<R>& x, promise<R>& y);
```

23 *Effects:* `x.swap(y)`.

30.194.1 Class template `future` [futures.unique_future]

1 The class template `future` defines a type for asynchronous return objects which do not share their shared state with other asynchronous return objects. A default-constructed `future` object has no shared state. A `future` object with shared state can be created by functions on asynchronous providers (30.179.4) or by the move constructor and shares its shared state with the original asynchronous provider. The result (value or exception) of a `future` object can be set by calling a respective function on an object that shares the same shared state.

2 [*Note:* Member functions of `future` do not synchronize with themselves or with member functions of `shared_future`. — *end note*]

3

The effect of calling any member function other than the destructor, the move-assignment operator, or `valid` on a `future` object for which `valid() == false` is undefined. [*Note:* Implementations are encouraged to detect this case and throw an object of type `future_error` with an error condition of `future_errc::no_state`. — *end note*]

```
namespace std {
  template <class R>
  class future {
  public:
    future() noexcept;
    future(future &&) noexcept;
    future(const future& rhs) = delete;
    ~future();
    future& operator=(const future& rhs) = delete;
    future& operator=(future&&) noexcept;
    shared_future<R> share();

    // retrieving the value
    see below get();

    // functions to check state
    bool valid() const noexcept;

    void wait() const;
    template <class Rep, class Period>
      future_status wait_for(const chrono::duration<Rep, Period>& rel_time) const;
    template <class Clock, class Duration>
      future_status wait_until(const chrono::time_point<Clock, Duration>& abs_time) const;
  };
}
```

1 The implementation shall provide the template `future` and two specializations, `future<R&>` and `future<void>`. These differ only in the return type and return value of the member function `get`, as set out in its description, below.

`future() noexcept;`

2 *Effects:* constructs an *empty* `future` object that does not refer to a shared state.

3 *Postcondition:* `valid() == false`.

`future(future&& rhs) noexcept;`

4 *Effects:* move constructs a `future` object that refers to the shared state that was originally referred to by `rhs` (if any).

5 *Postconditions:*

(5.1) — `valid()` returns the same value as `rhs.valid()` prior to the constructor invocation.

(5.2) — `rhs.valid() == false`.

`~future();`

6

Effects:

— releases any shared state (30.179.4);

— destroys `*this`.

```
future& operator=(future&& rhs) noexcept;
```

1

Effects:

— releases any shared state (30.179.4).

— move assigns the contents of `rhs` to `*this`.

1 *Postconditions:*

(1.1) — `valid()` returns the same value as `rhs.valid()` prior to the assignment.

(1.2) — `rhs.valid() == false`.

```
shared_future<R> share();
```

2 *Returns:* `shared_future<R>(std::move(*this))`.

3 *Postcondition:* `valid() == false`.

```
R future::get();
R& future<R&>::get();
void future<void>::get();
```

4 *Note:* as described above, the template and its two required specializations differ only in the return type and return value of the member function `get`.

5 *Effects:* `wait()`s until the shared state is ready, then retrieves the value stored in the shared state.

6 *Returns:*

(6.1) — `future::get()` returns the value `v` stored in the object's shared state as `std::move(v)`.

(6.2) — `future<R&>::get()` returns the reference stored as value in the object's shared state.

(6.3) — `future<void>::get()` returns nothing.

7

Throws: the stored exception, if an exception was stored in the shared state.

1

Postcondition: `valid() == false`.

```
bool valid() const noexcept;
```

1

Returns: `true` only if `*this` refers to a shared state.

```
void wait() const;
```

1 *Effects:* blocks until the shared state is ready.

```
template <class Rep, class Period>
  future_status wait_for(const chrono::duration<Rep, Period>& rel_time) const;
```

2 *Effects:* none if the shared state contains a deferred function (30.207.1), otherwise blocks until the shared state is ready or until the relative timeout (30.8.1) specified by `rel_time` has expired.

3 *Returns:*

(3.1) — `future_status::deferred` if the shared state contains a deferred function.

(3.2) — `future_status::ready` if the shared state is ready.

(3.3) — `future_status::timeout` if the function is returning because the relative timeout (30.8.1) specified by `rel_time` has expired.

4 *Throws:* timeout-related exceptions (30.8.1).

```
template <class Clock, class Duration>
  future_status wait_until(const chrono::time_point<Clock, Duration>& abs_time) const;
```

5 *Effects:* none if the shared state contains a deferred function (30.207.1), otherwise blocks until the shared state is ready or until the absolute timeout (30.8.1) specified by `abs_time` has expired.

6 *Returns:*

(6.1) — `future_status::deferred` if the shared state contains a deferred function.

(6.2) — `future_status::ready` if the shared state is ready.

(6.3) — `future_status::timeout` if the function is returning because the absolute timeout (30.8.1) specified by `abs_time` has expired.

7 *Throws:* timeout-related exceptions (30.8.1).

30.203.1 Class template `shared_future` [futures.shared_future]

1 The class template `shared_future` defines a type for asynchronous return objects which may share their shared state with other asynchronous return objects. A default-constructed `shared_future` object has no shared state. A `shared_future` object with shared state can be created by conversion from a `future` object and shares its shared state with the original asynchronous provider (30.179.4) of the shared state. The result (value or exception) of a `shared_future` object can be set by calling a respective function on an object that shares the same shared state.

2

[*Note:* Member functions of `shared_future` do not synchronize with themselves, but they synchronize with the shared shared state. — *end note*]

1 The effect of calling any member function other than the destructor, the move-assignment operator, or `valid()` on a `shared_future` object for which `valid() == false` is undefined. [*Note:* Implementations are encouraged to detect this case and throw an object of type `future_error` with an error condition of `future_errc::no_state`. — *end note*]

```
namespace std {
  template <class R>
  class shared_future {
  public:
    shared_future() noexcept;
    shared_future(const shared_future& rhs);
    shared_future(future<R>&&) noexcept;
    shared_future(shared_future&& rhs) noexcept;
    ~shared_future();
    shared_future& operator=(const shared_future& rhs);
    shared_future& operator=(shared_future&& rhs) noexcept;

    // retrieving the value
    see below get() const;

    // functions to check state
    bool valid() const noexcept;
```

```
    void wait() const;
    template <class Rep, class Period>
      future_status wait_for(const chrono::duration<Rep, Period>& rel_time) const;
    template <class Clock, class Duration>
      future_status wait_until(const chrono::time_point<Clock, Duration>& abs_time) const;
  };
}
```

2 The implementation shall provide the template `shared_future` and two specializations, `shared_future<R&>` and `shared_future<void>`. These differ only in the return type and return value of the member function `get`, as set out in its description, below.

`shared_future() noexcept;`

3 *Effects:* constructs an *empty* `shared_future` object that does not refer to a shared state.

4 *Postcondition:* `valid() == false`.

`shared_future(const shared_future& rhs);`

5 *Effects:* constructs a `shared_future` object that refers to the same shared state as `rhs` (if any).

6 *Postcondition:* `valid()` returns the same value as `rhs.valid()`.

```
shared_future(future<R>&& rhs) noexcept;
shared_future(shared_future&& rhs) noexcept;
```

7 *Effects:* move constructs a `shared_future` object that refers to the shared state that was originally referred to by `rhs` (if any).

8 *Postconditions:*

(8.1) — `valid()` returns the same value as `rhs.valid()` returned prior to the constructor invocation.

(8.2) — `rhs.valid() == false`.

`~shared_future();`

9

Effects:

— releases any shared state (30.179.4);

— destroys `*this`.

`shared_future& operator=(shared_future&& rhs) noexcept;`

1 *Effects:*

(1.1) — releases any shared state (30.179.4);

(1.2) — move assigns the contents of `rhs` to `*this`.

2 *Postconditions:*

(2.1) — `valid()` returns the same value as `rhs.valid()` returned prior to the assignment.

(2.2) — `rhs.valid() == false`.

`shared_future& operator=(const shared_future& rhs);`

3 *Effects:*

(3.1) — releases any shared state (30.179.4);

(3.2) — assigns the contents of `rhs` to `*this`. [*Note:* As a result, `*this` refers to the same shared state as `rhs` (if any). — *end note*]

4 *Postconditions:* `valid() == rhs.valid()`.

```
const R& shared_future::get() const;
R& shared_future<R&>::get() const;
void shared_future<void>::get() const;
```

5 *Note:* as described above, the template and its two required specializations differ only in the return type and return value of the member function `get`.

6

Note: access to a value object stored in the shared state is unsynchronized, so programmers should apply only those operations on `R` that do not introduce a data race (1.859).

1 *Effects:* `wait()`s until the shared state is ready, then retrieves the value stored in the shared state.

2 *Returns:*

(2.1) — `shared_future::get()` returns a const reference to the value stored in the object's shared state. [*Note:* Access through that reference after the shared state has been destroyed produces undefined behavior; this can be avoided by not storing the reference in any storage with a greater lifetime than the `shared_future` object that returned the reference. — *end note*]

(2.2) — `shared_future<R&>::get()` returns the reference stored as value in the object's shared state.

(2.3) — `shared_future<void>::get()` returns nothing.

3 *Throws:* the stored exception, if an exception was stored in the shared state.

```
bool valid() const noexcept;
```

4 *Returns:* `true` only if `*this` refers to a shared state.

```
void wait() const;
```

5 *Effects:* blocks until the shared state is ready.

```
template <class Rep, class Period>
  future_status wait_for(const chrono::duration<Rep, Period>& rel_time) const;
```

6 *Effects:* none if the shared state contains a deferred function (30.207.1), otherwise blocks until the shared state is ready or until the relative timeout (30.8.1) specified by `rel_time` has expired.

7 *Returns:*

(7.1) — `future_status::deferred` if the shared state contains a deferred function.

(7.2) — `future_status::ready` if the shared state is ready.

(7.3) — `future_status::timeout` if the function is returning because the relative timeout (30.8.1) specified by `rel_time` has expired.

8 *Throws:* timeout-related exceptions (30.8.1).

```
template <class Clock, class Duration>
  future_status wait_until(const chrono::time_point<Clock, Duration>& abs_time) const;
```

9 *Effects:* none if the shared state contains a deferred function (30.207.1), otherwise blocks until the shared state is ready or until the absolute timeout (30.8.1) specified by `abs_time` has expired.

10 *Returns:*

(10.1) — `future_status::deferred` if the shared state contains a deferred function.

(10.2) — `future_status::ready` if the shared state is ready.

(10.3) — `future_status::timeout` if the function is returning because the absolute timeout (30.8.1) specified by `abs_time` has expired.

11 *Throws:* timeout-related exceptions (30.8.1).

30.207.1 Function template `async` [futures.async]

1

The function template `async` provides a mechanism to launch a function potentially in a new thread and provides the result of the function in a `future` object with which it shares a shared state.

```
template <class F, class... Args>
  future<result_of_t<decay_t<F>(decay_t<Args>...)>> async(F&& f, Args&&... args);
template <class F, class... Args>
  future<result_of_t<decay_t<F>(decay_t<Args>...)>> async(launch policy, F&& f, Args&&... args);
```

1 *Requires:* `F` and each `Ti` in `Args` shall satisfy the `MoveConstructible` requirements. *`INVOKE`*`(`*`DECAY_COPY`*`(std::forward<F>(f)), `*`DECAY_COPY`*`(std::forward<Args>(args))...)` (20.295.1, 30.31.0.1) shall be a valid expression.

2

Effects: The first function behaves the same as a call to the second function with a `policy` argument of `launch::async | launch::deferred` and the same arguments for `F` and `Args`. The second function creates a shared state that is associated with the returned `future` object. The further behavior of the second function depends on the `policy` argument as follows (if more than one of these conditions applies, the implementation may choose any of the corresponding policies):

— if `policy & launch::async` is non-zero — calls *`INVOKE`*`(`*`DECAY_COPY`*`(std::forward<F>(f)), `*`DECAY_COPY`*`(std::forward<Args>(args))...)` (20.295.1, 30.31.0.1) as if in a new thread of execution represented by a `thread` object with the calls to *`DECAY_COPY`*`()` being evaluated in the thread that called `async`. Any return value is stored as the result in the shared state. Any exception propagated from the execution of *`INVOKE`*`(`*`DECAY_COPY`*`(std::forward<F>(f)), `*`DECAY_COPY`*`(std::forward<Args>(args))...)` is stored as the exceptional result in the shared state. The `thread` object is stored in the shared state and affects the behavior of any asynchronous return objects that reference that state.

— if `policy & launch::deferred` is non-zero — Stores *`DECAY_COPY`*`(std::forward<F>(f))` and *`DECAY_COPY`*`(std::forward<Args>(args))...` in the shared state. These copies of `f` and `args` constitute a *deferred function*. Invocation of the deferred function evaluates *`INVOKE`*`(std::move(g), std::move(xyz))` where `g` is the stored value of *`DECAY_COPY`*`(std::forward<F>(f))` and `xyz` is the stored copy of *`DECAY_COPY`*`(std::forward<Args>(args))...`. Any return value is stored as the result in the shared state. Any exception propagated from the execution of the deferred function is stored as the exceptional result in the shared state. The shared state is not made ready until the function has completed. The first call to a non-timed waiting function (30.179.4) on an asynchronous return object referring to this shared state shall invoke the deferred function in the thread that called the waiting function. Once evaluation of *`INVOKE`*`(std::move(g), std::move(xyz))` begins, the function is no longer considered deferred. [*Note:* If this policy is specified together with other policies, such as when using a `policy` value of `launch::async | launch::deferred`, implementations should defer invocation or the selection of the policy when no more concurrency can be effectively exploited. — *end note*]

— If no value is set in the launch policy, or a value is set that is neither specified in this International Standard or by the implementation, the behaviour is undefined.

1

Returns: An object of type `future<result_of_t<decay_t<F>(decay_t<Args>...)>>` that refers to the shared state created by this call to `async`. [*Note:* If a future obtained from std::async is moved outside the local scope, other code that uses the future must be aware that the future's destructor may block for the shared state to become ready. — *end note*]

1

Synchronization: Regardless of the provided `policy` argument,

— the invocation of `async` synchronizes with (1.859) the invocation of `f`. [*Note:* This statement applies even when the corresponding `future` object is moved to another thread. — *end note*]; and

— the completion of the function `f` is sequenced before (1.859) the shared state is made ready. [*Note:* `f` might not be called at all, so its completion might never happen. — *end note*]

If the implementation chooses the `launch::async` policy,

— a call to a waiting function on an asynchronous return object that shares the shared state created by this `async` call shall block until the associated thread has completed, as if joined, or else time out (30.75.0.1);

— the associated thread completion synchronizes with (1.859) the return from the first function that successfully detects the ready status of the shared state or with the return from the last function that releases the shared state, whichever happens first.

1

Throws: `system_error` if `policy == launch::async` and the implementation is unable to start a new thread.

1

Error conditions:

— `resource_unavailable_try_again` — if `policy == launch::async` and the system is unable to start a new thread.

1 [*Example:*

```
int work1(int value);
int work2(int value);
int work(int value) {
  auto handle = std::async([=]{ return work2(value); });
  int tmp = work1(value);
  return tmp + handle.get();    // #1
}
```

[*Note:* Line #1 might not result in concurrency because the `async` call uses the default policy, which may use `launch::deferred`, in which case the lambda might not be invoked until the `get()` call; in that case, `work1` and `work2` are called on the same thread and there is no concurrency. — *end note*] — *end example*]

30.238.1 Class template packaged_task [futures.task]

1

The class template packaged_task defines a type for wrapping a function or callable object so that the return value of the function or callable object is stored in a future when it is invoked.

1 When the packaged_task object is invoked, its stored task is invoked and the result (whether normal or exceptional) stored in the shared state. Any futures that share the shared state will then be able to access the stored result.

```
namespace std {
  template<class> class packaged_task; // undefined

  template<class R, class... ArgTypes>
  class packaged_task<R(ArgTypes...)> {
  public:
    // construction and destruction
    packaged_task() noexcept;
    template <class F>
      explicit packaged_task(F&& f);
    template <class F, class Allocator>
      explicit packaged_task(allocator_arg_t, const Allocator& a, F&& f);
    ~packaged_task();

    // no copy
    packaged_task(const packaged_task&) = delete;
    packaged_task& operator=(const packaged_task&) = delete;

    // move support
    packaged_task(packaged_task&& rhs) noexcept;
    packaged_task& operator=(packaged_task&& rhs) noexcept;
    void swap(packaged_task& other) noexcept;

    bool valid() const noexcept;

    // result retrieval
    future<R> get_future();

    // execution
    void operator()(ArgTypes... );
    void make_ready_at_thread_exit(ArgTypes...);

    void reset();
  };
  template <class R, class... ArgTypes>
    void swap(packaged_task<R(ArgTypes...)>& x, packaged_task<R(ArgTypes...)>& y) noexcept;
  template <class R, class Alloc>
    struct uses_allocator<packaged_task<R>, Alloc>;
}
```

30.239.0.1 packaged_task member functions [futures.task.members]

```
packaged_task() noexcept;
```

1 *Effects:* constructs a packaged_task object with no shared state and no stored task.

```
template <class F>
```

```
  packaged_task(F&& f);
template <class F, class Allocator>
  explicit packaged_task(allocator_arg_t, const Allocator& a, F&& f);
```

2

Requires: `INVOKE(f, t1, t2, ..., tN, R)`, where `t1, t2, ..., tN` are values of the corresponding types in `ArgTypes...`, shall be a valid expression. Invoking a copy of `f` shall behave the same as invoking `f`.

1 *Remarks:* These constructors shall not participate in overload resolution if `decay_t<F>` is the same type as `std::packaged_task<R(ArgTypes...)>`.

2

Effects: constructs a new `packaged_task` object with a shared state and initializes the object's stored task with `std::forward<F>(f)`. The constructors that take an `Allocator` argument use it to allocate memory needed to store the internal data structures.

1

Throws: any exceptions thrown by the copy or move constructor of `f`, or `std::bad_alloc` if memory for the internal data structures could not be allocated.

```
packaged_task(packaged_task&& rhs) noexcept;
```

1 *Effects:* constructs a new `packaged_task` object and transfers ownership of `rhs`'s shared state to `*this`, leaving `rhs` with no shared state. Moves the stored task from `rhs` to `*this`.

2 *Postcondition:* `rhs` has no shared state.

```
packaged_task& operator=(packaged_task&& rhs) noexcept;
```

3 *Effects:*

(3.1) — releases any shared state (30.179.4).

(3.2) — `packaged_task(std::move(rhs)).swap(*this)`.

```
~packaged_task();
```

4 *Effects:* Abandons any shared state. (30.179.4).

```
void swap(packaged_task& other) noexcept;
```

5 *Effects:* exchanges the shared states and stored tasks of `*this` and `other`.

6 *Postcondition:* `*this` has the same shared state and stored task (if any) as `other` prior to the call to `swap`. `other` has the same shared state and stored task (if any) as `*this` prior to the call to `swap`.

```
bool valid() const noexcept;
```

7 *Returns:* `true` only if `*this` has a shared state.

```
future<R> get_future();
```

8 *Returns:* A `future` object that shares the same shared state as `*this`.

9 *Throws:* a `future_error` object if an error occurs.

10

Error conditions:

— `future_already_retrieved` if `get_future` has already been called on a `packaged_task` object with the same shared state as `*this`.

— `no_state` if `*this` has no shared state.

```
void operator()(ArgTypes... args);
```

1 *Effects:* *INVOKE*`(f, t1, t2, ..., tN, R)`, where `f` is the stored task of `*this` and `t1, t2, ..., tN` are the values in `args...`. If the task returns normally, the return value is stored as the asynchronous result in the shared state of `*this`, otherwise the exception thrown by the task is stored. The shared state of `*this` is made ready, and any threads blocked in a function waiting for the shared state of `*this` to become ready are unblocked.

2 *Throws:* a `future_error` exception object if there is no shared state or the stored task has already been invoked.

3 *Error conditions:*

(3.1) — `promise_already_satisfied` if the stored task has already been invoked.

(3.2) — `no_state` if `*this` has no shared state.

```
void make_ready_at_thread_exit(ArgTypes... args);
```

4 *Effects:* *INVOKE*`(f, t1, t2, ..., tN, R)`, where `f` is the stored task and `t1, t2, ..., tN` are the values in `args...`. If the task returns normally, the return value is stored as the asynchronous result in the shared state of `*this`, otherwise the exception thrown by the task is stored. In either case, this shall be done without making that state ready (30.179.4) immediately. Schedules the shared state to be made ready when the current thread exits, after all objects of thread storage duration associated with the current thread have been destroyed.

5 *Throws:* `future_error` if an error condition occurs.

6 *Error conditions:*

(6.1) — `promise_already_satisfied` if the stored task has already been invoked.

(6.2) — `no_state` if `*this` has no shared state.

```
void reset();
```

7 *Effects:* as if `*this = packaged_task(std::move(f))`, where `f` is the task stored in `*this`. [*Note:* This constructs a new shared state for `*this`. The old state is abandoned (30.179.4). — *end note*]

1 *Throws:*

(1.1) — `bad_alloc` if memory for the new shared state could not be allocated.

(1.2) — any exception thrown by the move constructor of the task stored in the shared state.

(1.3) — `future_error` with an error condition of `no_state` if `*this` has no shared state.

30.247.0.1 `packaged_task` globals [futures.task.nonmembers]

```
template <class R, class... ArgTypes>
  void swap(packaged_task<R(ArgTypes...)>& x, packaged_task<R(ArgTypes...)>& y) noexcept;
```

1 *Effects:* `x.swap(y)`

```
template <class R, class Alloc>
  struct uses_allocator<packaged_task<R>, Alloc>
    : true_type { };
```

2 *Requires:* `Alloc` shall be an Allocator (17.165.0.1).

Annex A (informative) Grammar summary [gram]

1 This summary of C++ syntax is intended to be an aid to comprehension. It is not an exact statement of the language. In particular, the grammar described here accepts a superset of valid C++ constructs. Disambiguation rules (6.556, 7.40, 10.52) must be applied to distinguish expressions from declarations. Further, access control, ambiguity, and type rules must be used to weed out syntactically valid but meaningless constructs.

A.1 Keywords [gram.key]

1 New context-dependent keywords are introduced into a program by `typedef` (7.235.1), `namespace` (7.1149.1), class (Clause 9), enumeration (7.1001), and `template` (Clause 14) declarations.

typedef-name:
    identifier
namespace-name:
    original-namespace-name
    namespace-alias

original-namespace-name:
    identifier

namespace-alias:
    identifier

class-name:
    identifier
    simple-template-id
enum-name:
    identifier

template-name:
    identifier

Note that a *typedef-name* naming a class is also a *class-name* (9.172).

A.2 Lexical conventions [gram.lex]

hex-quad:
    hexadecimal-digit hexadecimal-digit hexadecimal-digit hexadecimal-digit

universal-character-name:
    `\u` *hex-quad*
    `\U` *hex-quad hex-quad*

preprocessing-token:
    header-name
    identifier
    pp-number
    character-literal
    user-defined-character-literal
    string-literal
    user-defined-string-literal
    preprocessing-op-or-punc
    each non-white-space character that cannot be one of the above

token:
- *identifier*
- *keyword*
- *literal*
- *operator*
- *punctuator*

header-name:
- `<` *h-char-sequence* `>`
- `"` *q-char-sequence* `"`

h-char-sequence:
- *h-char*
- *h-char-sequence h-char*

h-char:
- any member of the source character set except new-line and `>`

q-char-sequence:
- *q-char*
- *q-char-sequence q-char*

q-char:
- any member of the source character set except new-line and `"`

pp-number:
- *digit*
- `.` *digit*
- *pp-number digit*
- *pp-number identifier-nondigit*
- *pp-number* `'` *digit*
- *pp-number* `'` *nondigit*
- *pp-number* `e` *sign*
- *pp-number* `E` *sign*
- *pp-number* `.`

identifier:
- *identifier-nondigit*
- *identifier identifier-nondigit*
- *identifier digit*

identifier-nondigit:
- *nondigit*
- *universal-character-name*
- other implementation-defined characters

nondigit: one of

```
a b c d e f g h i j k l m
n o p q r s t u v w x y z
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z _
```

digit: one of

```
0 1 2 3 4 5 6 7 8 9
```

preprocessing-op-or-punc: one of

```
{     }         [         ]         #         ##        (         )
<:    :>        <%        %>        %:        %:%:      ;         :         ...
new   delete    ?         ::        .         .*
+     -         *         /         %         ^         &         |         ~
!     =         <         >         +=        -=        *=        /=        %=
^=    &=        |=        <<        >>        >>=       <<=       ==        !=
<=    >=        &&        ||        ++        --        ,         ->*       ->
and   and_eq    bitand    bitor     compl     not       not_eq
or    or_eq     xor       xor_eq
```

literal:
- *integer-literal*
- *character-literal*
- *floating-literal*
- *string-literal*
- *boolean-literal*
- *pointer-literal*
- *user-defined-literal*

integer-literal:
- *decimal-literal integer-suffix*$_{opt}$
- *octal-literal integer-suffix*$_{opt}$
- *hexadecimal-literal integer-suffix*$_{opt}$
- *binary-literal integer-suffix*$_{opt}$

decimal-literal:
- *nonzero-digit*
- *decimal-literal* '$_{opt}$ *digit*

octal-literal:
- `0`
- *octal-literal* '$_{opt}$ *octal-digit*

hexadecimal-literal:
- `0x` *hexadecimal-digit*
- `0X` *hexadecimal-digit*
- *hexadecimal-literal* '$_{opt}$ *hexadecimal-digit*

binary-literal:
- `0b` *binary-digit*
- `0B` *binary-digit*
- *binary-literal* '$_{opt}$ *binary-digit*

nonzero-digit: one of
`1 2 3 4 5 6 7 8 9`

octal-digit: one of
`0 1 2 3 4 5 6 7`

hexadecimal-digit: one of
`0 1 2 3 4 5 6 7 8 9`
`a b c d e f`
`A B C D E F`

binary-digit:
`0`
`1`

integer-suffix:
- *unsigned-suffix long-suffix*$_{opt}$
- *unsigned-suffix long-long-suffix*$_{opt}$
- *long-suffix unsigned-suffix*$_{opt}$
- *long-long-suffix unsigned-suffix*$_{opt}$

unsigned-suffix: one of
`u U`
long-suffix: one of
`l L`
long-long-suffix: one of
`ll LL`
character-literal:
`'` *c-char-sequence* `'`
`u'` *c-char-sequence* `'`
`U'` *c-char-sequence* `'`
`L'` *c-char-sequence* `'`
c-char-sequence:
c-char
c-char-sequence c-char
c-char:
any member of the source character set except
the single-quote `'`, backslash `\`, or new-line character
escape-sequence
universal-character-name
escape-sequence:
simple-escape-sequence
octal-escape-sequence
hexadecimal-escape-sequence
simple-escape-sequence: one of
`\'` `\"` `\?` `\\`
`\a` `\b` `\f` `\n` `\r` `\t` `\v`
octal-escape-sequence:
`\` *octal-digit*
`\` *octal-digit octal-digit*
`\` *octal-digit octal-digit octal-digit*
hexadecimal-escape-sequence:
`\x` *hexadecimal-digit*
hexadecimal-escape-sequence hexadecimal-digit
floating-literal:
fractional-constant exponent-part$_{opt}$ *floating-suffix*$_{opt}$
digit-sequence exponent-part floating-suffix$_{opt}$
fractional-constant:
digit-sequence$_{opt}$ `.` *digit-sequence*
digit-sequence `.`
exponent-part:
`e` *sign*$_{opt}$ *digit-sequence*
`E` *sign*$_{opt}$ *digit-sequence*
sign: one of
`+ -`
digit-sequence:
digit
digit-sequence `'`$_{opt}$ *digit*
floating-suffix: one of
`f l F L`
string-literal:
encoding-prefix$_{opt}$`"` *s-char-sequence*$_{opt}$`"`
encoding-prefix$_{opt}$`R` *raw-string*

encoding-prefix:
`u8`
`u`
`U`
`L`

s-char-sequence:
s-char
s-char-sequence s-char

s-char:
any member of the source character set except
the double-quote `"`, backslash `\`, or new-line character
escape-sequence
universal-character-name

raw-string:
`"` *d-char-sequence*$_{opt}$`(` *r-char-sequence*$_{opt}$`)` *d-char-sequence*$_{opt}$`"`

r-char-sequence:
r-char
r-char-sequence r-char

r-char:
any member of the source character set, except
a right parenthesis `)` followed by the initial *d-char-sequence*
(which may be empty) followed by a double quote `"`.

d-char-sequence:
d-char
d-char-sequence d-char

d-char:
any member of the basic source character set except:
space, the left parenthesis `(`, the right parenthesis `)`, the backslash `\`,
and the control characters representing horizontal tab,
vertical tab, form feed, and newline.

boolean-literal:
`false`
`true`

pointer-literal:
`nullptr`

user-defined-literal:
user-defined-integer-literal
user-defined-floating-literal
user-defined-string-literal
user-defined-character-literal

user-defined-integer-literal:
decimal-literal ud-suffix
octal-literal ud-suffix
hexadecimal-literal ud-suffix
binary-literal ud-suffix

user-defined-floating-literal:
fractional-constant exponent-part$_{opt}$ *ud-suffix*
digit-sequence exponent-part ud-suffix

user-defined-string-literal:
string-literal ud-suffix

user-defined-character-literal:
character-literal ud-suffix

ud-suffix:
  identifier

A.3 Basic concepts [gram.basic]

translation-unit:
  declaration-seq$_{opt}$

A.4 Expressions [gram.expr]

primary-expression:
  literal
  `this`
  `(` *expression* `)`
  id-expression
  lambda-expression

id-expression:
  unqualified-id
  qualified-id

unqualified-id:
  identifier
  operator-function-id
  conversion-function-id
  literal-operator-id
  `~` *class-name*
  `~` *decltype-specifier*
  template-id

qualified-id:
  nested-name-specifier `template`$_{opt}$ *unqualified-id*

nested-name-specifier:
  `::`
  type-name `::`
  namespace-name `::`
  decltype-specifier `::`
  nested-name-specifier identifier `::`
  nested-name-specifier `template`$_{opt}$ *simple-template-id* `::`

lambda-expression:
  lambda-introducer lambda-declarator$_{opt}$ *compound-statement*

lambda-introducer:
  `[` *lambda-capture*$_{opt}$`]`

lambda-capture:
  capture-default
  capture-list
  capture-default `,` *capture-list*

capture-default:
  `&`
  `=`

capture-list:
  capture `...`$_{opt}$
  capture-list `,` *capture* `...`$_{opt}$

capture:
  simple-capture
  init-capture

simple-capture:
  identifier
  `&` *identifier*
  `this`
init-capture:
  identifier initializer
  `&` *identifier initializer*
lambda-declarator:
  `(` *parameter-declaration-clause* `)` `mutable`$_{opt}$
    exception-specification$_{opt}$ *attribute-specifier-seq*$_{opt}$ *trailing-return-type*$_{opt}$
postfix-expression:
  primary-expression
  postfix-expression `[` *expression* `]`
  postfix-expression `[` *braced-init-list* `]`
  postfix-expression `(` *expression-list*$_{opt}$`)`
  simple-type-specifier `(` *expression-list*$_{opt}$`)`
  typename-specifier `(` *expression-list*$_{opt}$`)`
  simple-type-specifier braced-init-list
  typename-specifier braced-init-list
  postfix-expression `.` `template`$_{opt}$ *id-expression*
  postfix-expression `->` `template`$_{opt}$ *id-expression*
  postfix-expression `.` *pseudo-destructor-name*
  postfix-expression `->` *pseudo-destructor-name*
  postfix-expression `++`
  postfix-expression `--`
  `dynamic_cast` `<` *type-id* `>` `(` *expression* `)`
  `static_cast` `<` *type-id* `>` `(` *expression* `)`
  `reinterpret_cast` `<` *type-id* `>` `(` *expression* `)`
  `const_cast` `<` *type-id* `>` `(` *expression* `)`
  `typeid` `(` *expression* `)`
  `typeid` `(` *type-id* `)`
expression-list:
  initializer-list
pseudo-destructor-name:
  nested-name-specifier$_{opt}$ *type-name* `::` `~` *type-name*
  nested-name-specifier `template` *simple-template-id* `::` `~` *type-name*
  nested-name-specifier$_{opt}$`~` *type-name*
  `~` *decltype-specifier*
unary-expression:
  postfix-expression
  `++` *cast-expression*
  `--` *cast-expression*
  unary-operator cast-expression
  `sizeof` *unary-expression*
  `sizeof` `(` *type-id* `)`
  `sizeof` `...` `(` *identifier* `)`
  `alignof` `(` *type-id* `)`
  noexcept-expression
  new-expression
  delete-expression
unary-operator: one of
  `*` `&` `+` `-` `!` `~`
new-expression:
  `::`$_{opt}$`new` *new-placement*$_{opt}$ *new-type-id new-initializer*$_{opt}$
  `::`$_{opt}$`new` *new-placement*$_{opt}$`(` *type-id* `)` *new-initializer*$_{opt}$

new-placement:
`(` *expression-list* `)`

new-type-id:
type-specifier-seq new-declarator$_{opt}$

new-declarator:
ptr-operator new-declarator$_{opt}$
noptr-new-declarator

noptr-new-declarator:
`[` *expression* `]` *attribute-specifier-seq*$_{opt}$
noptr-new-declarator `[` *constant-expression* `]` *attribute-specifier-seq*$_{opt}$

new-initializer:
`(` *expression-list*$_{opt}$`)`
braced-init-list

delete-expression:
`::`$_{opt}$`delete` *cast-expression*
`::`$_{opt}$`delete [ ]` *cast-expression*

noexcept-expression:
`noexcept (` *expression* `)`

cast-expression:
unary-expression
`(` *type-id* `)` *cast-expression*

pm-expression:
cast-expression
pm-expression `.*` *cast-expression*
pm-expression `->*` *cast-expression*

multiplicative-expression:
pm-expression
multiplicative-expression `*` *pm-expression*
multiplicative-expression `/` *pm-expression*
multiplicative-expression `%` *pm-expression*

additive-expression:
multiplicative-expression
additive-expression `+` *multiplicative-expression*
additive-expression `-` *multiplicative-expression*

shift-expression:
additive-expression
shift-expression `<<` *additive-expression*
shift-expression `>>` *additive-expression*

relational-expression:
shift-expression
relational-expression `<` *shift-expression*
relational-expression `>` *shift-expression*
relational-expression `<=` *shift-expression*
relational-expression `>=` *shift-expression*

equality-expression:
relational-expression
equality-expression `==` *relational-expression*
equality-expression `!=` *relational-expression*

and-expression:
equality-expression
and-expression `&` *equality-expression*

exclusive-or-expression:
  and-expression
  exclusive-or-expression `^` *and-expression*

inclusive-or-expression:
  exclusive-or-expression
  inclusive-or-expression `|` *exclusive-or-expression*

logical-and-expression:
  inclusive-or-expression
  logical-and-expression `&&` *inclusive-or-expression*

logical-or-expression:
  logical-and-expression
  logical-or-expression || *logical-and-expression*

conditional-expression:
  logical-or-expression
  logical-or-expression `?` *expression* `:` *assignment-expression*

assignment-expression:
  conditional-expression
  logical-or-expression assignment-operator initializer-clause
  throw-expression

assignment-operator: one of
  `= *= /= %= += -= >>= <<= &= ^= |=`

expression:
  assignment-expression
  expression `,` *assignment-expression*

constant-expression:
  conditional-expression

A.5 Statements [gram.stmt]

statement:
  labeled-statement
  attribute-specifier-seq$_{opt}$ *expression-statement*
  attribute-specifier-seq$_{opt}$ *compound-statement*
  attribute-specifier-seq$_{opt}$ *selection-statement*
  attribute-specifier-seq$_{opt}$ *iteration-statement*
  attribute-specifier-seq$_{opt}$ *jump-statement*
  declaration-statement
  attribute-specifier-seq$_{opt}$ *try-block*

labeled-statement:
  attribute-specifier-seq$_{opt}$ *identifier* `:` *statement*
  attribute-specifier-seq$_{opt}$`case` *constant-expression* `:` *statement*
  attribute-specifier-seq$_{opt}$`default` `:` *statement*

expression-statement:
  expression$_{opt}$`;`

compound-statement:
  `{` *statement-seq*$_{opt}$`}`

statement-seq:
  statement
  statement-seq statement

selection-statement:
  `if` `(` *condition* `)` *statement*
  `if` `(` *condition* `)` *statement* `else` *statement*
  `switch` `(` *condition* `)` *statement*

condition:
  expression
  attribute-specifier-seq$_{opt}$ *decl-specifier-seq declarator* `=` *initializer-clause*
  attribute-specifier-seq$_{opt}$ *decl-specifier-seq declarator braced-init-list*

iteration-statement:
  `while` `(` *condition* `)` *statement*
  `do` *statement* `while` `(` *expression* `)` `;`
  `for` `(` *for-init-statement condition*$_{opt}$`;` *expression*$_{opt}$`)` *statement*
  `for` `(` *for-range-declaration* `:` *for-range-initializer* `)` *statement*

for-init-statement:
  expression-statement
  simple-declaration

for-range-declaration:
  attribute-specifier-seq$_{opt}$ *decl-specifier-seq declarator*

for-range-initializer:
  expression
  braced-init-list

jump-statement:
  `break` `;`
  `continue` `;`
  `return` *expression*$_{opt}$`;`
  `return` *braced-init-list* `;`
  `goto` *identifier* `;`

declaration-statement:
  block-declaration

A.6 Declarations [gram.dcl]

declaration-seq:
  declaration
  declaration-seq declaration

declaration:
  block-declaration
  function-definition
  template-declaration
  explicit-instantiation
  explicit-specialization
  linkage-specification
  namespace-definition
  empty-declaration
  attribute-declaration

block-declaration:
  simple-declaration
  asm-definition
  namespace-alias-definition
  using-declaration
  using-directive
  static_assert-declaration
  alias-declaration
  opaque-enum-declaration

alias-declaration:
  `using` *identifier attribute-specifier-seq*$_{opt}$`=` *type-id* `;`

simple-declaration:
  decl-specifier-seq$_{opt}$ *init-declarator-list*$_{opt}$`;`
  attribute-specifier-seq decl-specifier-seq$_{opt}$ *init-declarator-list* `;`

static_assert-declaration:
`static_assert` (*constant-expression* , *string-literal*) ;

empty-declaration:
;

attribute-declaration:
attribute-specifier-seq ;

decl-specifier:
storage-class-specifier
type-specifier
function-specifier
`friend`
`typedef`
`constexpr`

decl-specifier-seq:
decl-specifier attribute-specifier-seq$_{opt}$
decl-specifier decl-specifier-seq

storage-class-specifier:
`register`
`static`
`thread_local`
`extern`
`mutable`

function-specifier:
`inline`
`virtual`
`explicit`

typedef-name:
identifier

type-specifier:
trailing-type-specifier
class-specifier
enum-specifier

trailing-type-specifier:
simple-type-specifier
elaborated-type-specifier
typename-specifier
cv-qualifier

type-specifier-seq:
type-specifier attribute-specifier-seq$_{opt}$
type-specifier type-specifier-seq

trailing-type-specifier-seq:
trailing-type-specifier attribute-specifier-seq$_{opt}$
trailing-type-specifier trailing-type-specifier-seq

simple-type-specifier:
  nested-name-specifier$_{opt}$ *type-name*
  nested-name-specifier `template` *simple-template-id*
  `char`
  `char16_t`
  `char32_t`
  `wchar_t`
  `bool`
  `short`
  `int`
  `long`
  `signed`
  `unsigned`
  `float`
  `double`
  `void`
  `auto`
  decltype-specifier

type-name:
  class-name
  enum-name
  typedef-name
  simple-template-id

decltype-specifier:
  `decltype (` *expression* `)`
  `decltype ( auto )`

elaborated-type-specifier:
  class-key attribute-specifier-seq$_{opt}$ *nested-name-specifier*$_{opt}$ *identifier*
  class-key simple-template-id
  class-key nested-name-specifier `template`$_{opt}$ *simple-template-id*
  `enum` *nested-name-specifier*$_{opt}$ *identifier*

enum-name:
  identifier

enum-specifier:
  enum-head `{` *enumerator-list*$_{opt}$`}`
  enum-head `{` *enumerator-list* `, }`

enum-head:
  enum-key attribute-specifier-seq$_{opt}$ *identifier*$_{opt}$ *enum-base*$_{opt}$
  enum-key attribute-specifier-seq$_{opt}$ *nested-name-specifier identifier*
    enum-base$_{opt}$

opaque-enum-declaration:
  enum-key attribute-specifier-seq$_{opt}$ *identifier enum-base*$_{opt}$ `;`

enum-key:
  `enum`
  `enum class`
  `enum struct`

enum-base:
  `:` *type-specifier-seq*

enumerator-list:
  enumerator-definition
  enumerator-list `,` *enumerator-definition*

enumerator-definition:
  enumerator
  enumerator `=` *constant-expression*

enumerator:
  identifier
namespace-name:
  original-namespace-name
  namespace-alias
original-namespace-name:
  identifier
namespace-definition:
  named-namespace-definition
  unnamed-namespace-definition
named-namespace-definition:
  original-namespace-definition
  extension-namespace-definition
original-namespace-definition:
  `inline`$_{opt}$ `namespace` *identifier* `{` *namespace-body* `}`
extension-namespace-definition:
  `inline`$_{opt}$ `namespace` *original-namespace-name* `{` *namespace-body* `}`
unnamed-namespace-definition:
  `inline`$_{opt}$ `namespace` `{` *namespace-body* `}`
namespace-body:
  declaration-seq$_{opt}$
namespace-alias:
  identifier
namespace-alias-definition:
  `namespace` *identifier* `=` *qualified-namespace-specifier* `;`
qualified-namespace-specifier:
  nested-name-specifier$_{opt}$ *namespace-name*
using-declaration:
  `using` `typename`$_{opt}$ *nested-name-specifier unqualified-id* `;`
  `using` `::` *unqualified-id* `;`
using-directive:
  attribute-specifier-seq$_{opt}$`using` `namespace` *nested-name-specifier*$_{opt}$ *namespace-name* `;`
asm-definition:
  `asm` `(` *string-literal* `)` `;`
linkage-specification:
  `extern` *string-literal* `{` *declaration-seq*$_{opt}$`}`
  `extern` *string-literal declaration*
attribute-specifier-seq:
  attribute-specifier-seq$_{opt}$ *attribute-specifier*
attribute-specifier:
  `[` `[` *attribute-list* `]` `]`
  alignment-specifier
alignment-specifier:
  `alignas` `(` *type-id* `...`$_{opt}$`)`
  `alignas` `(` *constant-expression* `...`$_{opt}$`)`
attribute-list:
  attribute$_{opt}$
  attribute-list `,` *attribute*$_{opt}$
  attribute `...`
  attribute-list `,` *attribute* `...`
attribute:
  attribute-token attribute-argument-clause$_{opt}$

attribute-token:
  identifier
  attribute-scoped-token

attribute-scoped-token:
  attribute-namespace `::` *identifier*

attribute-namespace:
  identifier

attribute-argument-clause:
  `(` *balanced-token-seq* `)`

balanced-token-seq:
  balanced-token$_{opt}$
  balanced-token-seq balanced-token

balanced-token:
  `(` *balanced-token-seq* `)`
  `[` *balanced-token-seq* `]`
  `{` *balanced-token-seq* `}`
  any *token* other than a parenthesis, a bracket, or a brace

A.7 Declarators [gram.decl]

init-declarator-list:
  init-declarator
  init-declarator-list `,` *init-declarator*

init-declarator:
  declarator initializer$_{opt}$

declarator:
  ptr-declarator
  noptr-declarator parameters-and-qualifiers trailing-return-type

ptr-declarator:
  noptr-declarator
  ptr-operator ptr-declarator

noptr-declarator:
  declarator-id attribute-specifier-seq$_{opt}$
  noptr-declarator parameters-and-qualifiers
  noptr-declarator `[` *constant-expression*$_{opt}$`]` *attribute-specifier-seq*$_{opt}$
  `(` *ptr-declarator* `)`

parameters-and-qualifiers:
  `(` *parameter-declaration-clause* `)` *cv-qualifier-seq*$_{opt}$
    ref-qualifier$_{opt}$ *exception-specification*$_{opt}$ *attribute-specifier-seq*$_{opt}$

trailing-return-type:
  `->` *trailing-type-specifier-seq abstract-declarator*$_{opt}$

ptr-operator:
  `*` *attribute-specifier-seq*$_{opt}$ *cv-qualifier-seq*$_{opt}$
  `&` *attribute-specifier-seq*$_{opt}$
  `&&` *attribute-specifier-seq*$_{opt}$
  nested-name-specifier `*` *attribute-specifier-seq*$_{opt}$ *cv-qualifier-seq*$_{opt}$

cv-qualifier-seq:
  cv-qualifier cv-qualifier-seq$_{opt}$

cv-qualifier:
  `const`
  `volatile`

ref-qualifier:
  `&`
  `&&`

declarator-id:
  ...$_{opt}$ *id-expression*

type-id:
  type-specifier-seq abstract-declarator$_{opt}$

abstract-declarator:
  ptr-abstract-declarator
  noptr-abstract-declarator$_{opt}$ *parameters-and-qualifiers trailing-return-type*
  abstract-pack-declarator

ptr-abstract-declarator:
  noptr-abstract-declarator
  ptr-operator ptr-abstract-declarator$_{opt}$

noptr-abstract-declarator:
  noptr-abstract-declarator$_{opt}$ *parameters-and-qualifiers*
  noptr-abstract-declarator$_{opt}$ `[` *constant-expression*$_{opt}$ `]` *attribute-specifier-seq*$_{opt}$
  `(` *ptr-abstract-declarator* `)`

abstract-pack-declarator:
  noptr-abstract-pack-declarator
  ptr-operator abstract-pack-declarator

noptr-abstract-pack-declarator:
  noptr-abstract-pack-declarator parameters-and-qualifiers
  noptr-abstract-pack-declarator `[` *constant-expression*$_{opt}$ `]` *attribute-specifier-seq*$_{opt}$
  ...

parameter-declaration-clause:
  parameter-declaration-list$_{opt}$...$_{opt}$
  parameter-declaration-list `,` ...

parameter-declaration-list:
  parameter-declaration
  parameter-declaration-list `,` *parameter-declaration*

parameter-declaration:
  attribute-specifier-seq$_{opt}$ *decl-specifier-seq declarator*
  attribute-specifier-seq$_{opt}$ *decl-specifier-seq declarator* `=` *initializer-clause*
  attribute-specifier-seq$_{opt}$ *decl-specifier-seq abstract-declarator*$_{opt}$
  attribute-specifier-seq$_{opt}$ *decl-specifier-seq abstract-declarator*$_{opt}$`=` *initializer-clause*

function-definition:
  attribute-specifier-seq$_{opt}$ *decl-specifier-seq*$_{opt}$ *declarator virt-specifier-seq*$_{opt}$ *function-body*

function-body:
  ctor-initializer$_{opt}$ *compound-statement*
  function-try-block
  `= default ;`
  `= delete ;`

initializer:
  brace-or-equal-initializer
  `(` *expression-list* `)`

brace-or-equal-initializer:
  `=` *initializer-clause*
  braced-init-list

initializer-clause:
  assignment-expression
  braced-init-list

initializer-list:
  initializer-clause ...$_{opt}$
  initializer-list `,` *initializer-clause* ...$_{opt}$

braced-init-list:
`{` *initializer-list* `,`$_{opt}$ `}`
`{ }`

A.8 Classes [gram.class]

class-name:
identifier
simple-template-id

class-specifier:
class-head `{` *member-specification*$_{opt}$`}`

class-head:
class-key attribute-specifier-seq$_{opt}$ *class-head-name class-virt-specifier*$_{opt}$ *base-clause*$_{opt}$
class-key attribute-specifier-seq$_{opt}$ *base-clause*$_{opt}$

class-head-name:
nested-name-specifier$_{opt}$ *class-name*

class-virt-specifier:
`final`

class-key:
`class`
`struct`
`union`

member-specification:
member-declaration member-specification$_{opt}$
access-specifier `:` *member-specification*$_{opt}$

member-declaration:
attribute-specifier-seq$_{opt}$ *decl-specifier-seq*$_{opt}$ *member-declarator-list*$_{opt}$ `;`
function-definition
using-declaration
static_assert-declaration
template-declaration
alias-declaration
empty-declaration

member-declarator-list:
member-declarator
member-declarator-list `,` *member-declarator*

member-declarator:
declarator virt-specifier-seq$_{opt}$ *pure-specifier*$_{opt}$
declarator brace-or-equal-initializer$_{opt}$
identifier$_{opt}$ *attribute-specifier-seq*$_{opt}$`:` *constant-expression*

virt-specifier-seq:
virt-specifier
virt-specifier-seq virt-specifier

virt-specifier:
`override`
`final`

pure-specifier:
`= 0`

A.9 Derived classes [gram.derived]

base-clause:
`:` *base-specifier-list*

base-specifier-list:
base-specifier `...`$_{opt}$
base-specifier-list `,` *base-specifier* `...`$_{opt}$

base-specifier:
  attribute-specifier-seq$_{opt}$ *base-type-specifier*
  attribute-specifier-seq$_{opt}$ `virtual` *access-specifier*$_{opt}$ *base-type-specifier*
  attribute-specifier-seq$_{opt}$ *access-specifier* `virtual`$_{opt}$ *base-type-specifier*

class-or-decltype:
  nested-name-specifier$_{opt}$ *class-name*
  decltype-specifier

base-type-specifier:
  class-or-decltype

access-specifier:
  `private`
  `protected`
  `public`

A.10 Special member functions [gram.special]

conversion-function-id:
  `operator` *conversion-type-id*

conversion-type-id:
  type-specifier-seq conversion-declarator$_{opt}$

conversion-declarator:
  ptr-operator conversion-declarator$_{opt}$

ctor-initializer:
  `:` *mem-initializer-list*

mem-initializer-list:
  mem-initializer `...`$_{opt}$
  mem-initializer `...`$_{opt}$ `,` *mem-initializer-list*

mem-initializer:
  mem-initializer-id `(` *expression-list*$_{opt}$`)`
  mem-initializer-id braced-init-list

mem-initializer-id:
  class-or-decltype
  identifier

A.11 Overloading [gram.over]

operator-function-id:
  `operator` *operator*

operator: one of

```
new   delete   new[]   delete[]
+     -        *       /        %        ^        &        |        ~
!     =        <       >        +=       -=       *=       /=       %=
^=    &=       |=      <<       >>       >>=      <<=      ==       !=
<=    >=       &&      ||       ++       --       ,        ->*      ->
()    []
```

literal-operator-id:
  `operator` *string-literal identifier*
  `operator` *user-defined-string-literal*

A.12 Templates [gram.temp]

template-declaration:
  `template <` *template-parameter-list* `>` *declaration*

template-parameter-list:
  template-parameter
  template-parameter-list `,` *template-parameter*

template-parameter:
  type-parameter
  parameter-declaration

type-parameter:
  `class` `...`$_{opt}$ *identifier*$_{opt}$
  `class` *identifier*$_{opt}$`=` *type-id*
  `typename` `...`$_{opt}$ *identifier*$_{opt}$
  `typename` *identifier*$_{opt}$`=` *type-id*
  `template` `<` *template-parameter-list* `>` `class` `...`$_{opt}$ *identifier*$_{opt}$
  `template` `<` *template-parameter-list* `>` `class` *identifier*$_{opt}$`=` *id-expression*

simple-template-id:
  template-name `<` *template-argument-list*$_{opt}$`>`

template-id:
  simple-template-id
  operator-function-id `<` *template-argument-list*$_{opt}$`>`
  literal-operator-id `<` *template-argument-list*$_{opt}$`>`

template-name:
  identifier

template-argument-list:
  template-argument `...`$_{opt}$
  template-argument-list `,` *template-argument* `...`$_{opt}$

template-argument:
  constant-expression
  type-id
  id-expression

typename-specifier:
  `typename` *nested-name-specifier identifier*
  `typename` *nested-name-specifier* `template`$_{opt}$ *simple-template-id*

explicit-instantiation:
  `extern`$_{opt}$ `template` *declaration*

explicit-specialization:
  `template` `<` `>` *declaration*

A.13 Exception handling [gram.except]

try-block:
  `try` *compound-statement handler-seq*

function-try-block:
  `try` *ctor-initializer*$_{opt}$ *compound-statement handler-seq*

handler-seq:
  handler handler-seq$_{opt}$

handler:
  `catch` `(` *exception-declaration* `)` *compound-statement*

exception-declaration:
  attribute-specifier-seq$_{opt}$ *type-specifier-seq declarator*
  attribute-specifier-seq$_{opt}$ *type-specifier-seq abstract-declarator*$_{opt}$
  `...`

throw-expression:
  `throw` *assignment-expression*$_{opt}$

exception-specification:
  dynamic-exception-specification
  noexcept-specification

dynamic-exception-specification:
`throw` (*type-id-list*$_{opt}$)

type-id-list:
type-id ...$_{opt}$
type-id-list , *type-id* ...$_{opt}$

noexcept-specification:
`noexcept` (*constant-expression*)
`noexcept`

A.14 Preprocessing directives [gram.cpp]

preprocessing-file:
group$_{opt}$

group:
group-part
group group-part

group-part:
if-section
control-line
text-line
non-directive

if-section:
if-group elif-groups$_{opt}$ *else-group*$_{opt}$ *endif-line*

if-group:
`if` *constant-expression new-line group*$_{opt}$
`ifdef` *identifier new-line group*$_{opt}$
`ifndef` *identifier new-line group*$_{opt}$

elif-groups:
elif-group
elif-groups elif-group

elif-group:
`elif` *constant-expression new-line group*$_{opt}$

else-group:
`else` *new-line group*$_{opt}$

endif-line:
`endif` *new-line*

control-line:
`include` *pp-tokens new-line*
`define` *identifier replacement-list new-line*
`define` *identifier lparen identifier-list*$_{opt}$) *replacement-list new-line*
`define` *identifier lparen* ...) *replacement-list new-line*
`define` *identifier lparen identifier-list,* ...) *replacement-list new-line*
`undef` *identifier new-line*
`line` *pp-tokens new-line*
`error` *pp-tokens*$_{opt}$ *new-line*
`pragma` *pp-tokens*$_{opt}$ *new-line*
new-line

text-line:
pp-tokens$_{opt}$ *new-line*

non-directive:
pp-tokens new-line

lparen:
a (character not immediately preceded by white-space

identifier-list:
  identifier
  identifier-list **,** *identifier*

replacement-list:
  pp-tokens$_{opt}$

pp-tokens:
  preprocessing-token
  pp-tokens preprocessing-token

new-line:
  the new-line character

Annex B (informative) Implementation quantities [implimits]

1

Because computers are finite, C++ implementations are inevitably limited in the size of the programs they can successfully process. Every implementation shall document those limitations where known. This documentation may cite fixed limits where they exist, say how to compute variable limits as a function of available resources, or say that fixed limits do not exist or are unknown.

1

The limits may constrain quantities that include those described below or others. The bracketed number following each quantity is recommended as the minimum for that quantity. However, these quantities are only guidelines and do not determine compliance.

— Nesting levels of compound statements, iteration control structures, and selection control structures [256].

— Nesting levels of conditional inclusion [256].

— Pointer, array, and function declarators (in any combination) modifying a class, arithmetic, or incomplete type in a declaration [256].

— Nesting levels of parenthesized expressions within a full-expression [256].

— Number of characters in an internal identifier or macro name [1 024].

— Number of characters in an external identifier [1 024].

— External identifiers in one translation unit [65 536].

— Identifiers with block scope declared in one block [1 024].

— Macro identifiers simultaneously defined in one translation unit [65 536].

— Parameters in one function definition [256].

— Arguments in one function call [256].

— Parameters in one macro definition [256].

— Arguments in one macro invocation [256].

— Characters in one logical source line [65 536].

— Characters in a string literal (after concatenation) [65 536].

— Size of an object [262 144].

— Nesting levels for `#include` files [256].

— Case labels for a `switch` statement (excluding those for any nested `switch` statements) [16 384].

— Data members in a single class [16 384].

— Enumeration constants in a single enumeration [4 096].

— Levels of nested class definitions in a single *member-specification* [256].

— Functions registered by `atexit()` [32].

— Functions registered by `at_quick_exit()` [32].

— Direct and indirect base classes [16 384].

— Direct base classes for a single class [1 024].

— Members declared in a single class [4 096].

— Final overriding virtual functions in a class, accessible or not [16 384].

— Direct and indirect virtual bases of a class [1 024].

— Static members of a class [1 024].

— Friend declarations in a class [4 096].

— Access control declarations in a class [4 096].

— Member initializers in a constructor definition [6 144].

— Scope qualifications of one identifier [256].

— Nested external specifications [1 024].

— Recursive `constexpr` function invocations [512].

— Full-expressions evaluated within a core constant expression [1 048 576].

— Template arguments in a template declaration [1 024].

— Recursively nested template instantiations, including substitution during template argument deduction (14.2114.1) [1 024].

— Handlers per `try` block [256].

— Throw specifications on a single function declaration [256].

— Number of placeholders (20.379.0.1) [10].

Annex C (informative) Compatibility [diff]

C.1 C++ and ISO C [diff.iso]

1 This subclause lists the differences between C++ and ISO C, by the chapters of this document.

C.1.1 Clause 2: lexical conventions [diff.lex]

2.287
Change: New Keywords
New keywords are added to C++; see 2.287.
Rationale: These keywords were added in order to implement the new semantics of C++.
Effect on original feature: Change to semantics of well-defined feature. Any ISO C programs that used any of these keywords as identifiers are not valid C++ programs.
Difficulty of converting: Syntactic transformation. Converting one specific program is easy. Converting a large collection of related programs takes more work.
How widely used: Common.

2.356.1
Change: Type of character literal is changed from `int` to `char`
Rationale: This is needed for improved overloaded function argument type matching. For example:

```
int function( int i );
int function( char c );

function( 'x' );
```

It is preferable that this call match the second version of function rather than the first.
Effect on original feature: Change to semantics of well-defined feature. ISO C programs which depend on

```
sizeof('x') == sizeof(int)
```

will not work the same as C++ programs.
Difficulty of converting: Simple.
How widely used: Programs which depend upon `sizeof('x')` are probably rare.

Subclause 2.453.1:
Change: String literals made const
The type of a string literal is changed from "array of `char`" to "array of `const char`." The type of a `char16_t` string literal is changed from "array of *some-integer-type*" to "array of `const char16_t`." The type of a `char32_t` string literal is changed from "array of *some-integer-type*" to "array of `const char32_t`." The type of a wide string literal is changed from "array of `wchar_t`" to "array of `const wchar_t`."
Rationale: This avoids calling an inappropriate overloaded function, which might expect to be able to modify its argument.
Effect on original feature: Change to semantics of well-defined feature.
Difficulty of converting: Syntactic transformation. The fix is to add a cast:

```
char* p = "abc";              // valid in C, invalid in C++
void f(char*) {
  char* p = (char*)"abc";     // OK: cast added
```

```
  f(p);
  f((char*)"def");              // OK: cast added
}
```

How widely used: Programs that have a legitimate reason to treat string literals as pointers to potentially modifiable memory are probably rare.

C.1.2 Clause 3: basic concepts [diff.basic]

3.43
Change: C++ does not have "tentative definitions" as in C E.g., at file scope,

```
int i;
int i;
```

is valid in C, invalid in C++. This makes it impossible to define mutually referential file-local static objects, if initializers are restricted to the syntactic forms of C. For example,

```
struct X { int i; struct X* next; };

static struct X a;
static struct X b = { 0, &a };
static struct X a = { 1, &b };
```

Rationale: This avoids having different initialization rules for fundamental types and user-defined types.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Semantic transformation.
Rationale: In C++, the initializer for one of a set of mutually-referential file-local static objects must invoke a function call to achieve the initialization.
How widely used: Seldom.

3.464
Change: A `struct` is a scope in C++, not in C
Rationale: Class scope is crucial to C++, and a struct is a class.
Effect on original feature: Change to semantics of well-defined feature.
Difficulty of converting: Semantic transformation.
How widely used: C programs use `struct` extremely frequently, but the change is only noticeable when `struct`, enumeration, or enumerator names are referred to outside the `struct`. The latter is probably rare.

3.1459 [also 7.519.1]
Change: A name of file scope that is explicitly declared `const`, and not explicitly declared `extern`, has internal linkage, while in C it would have external linkage
Rationale: Because `const` objects can be used as compile-time values in C++, this feature urges programmers to provide explicit initializer values for each `const`. This feature allows the user to put `const`objects in header files that are included in many compilation units.
Effect on original feature: Change to semantics of well-defined feature.
Difficulty of converting: Semantic transformation
How widely used: Seldom

3.1721
Change: Main cannot be called recursively and cannot have its address taken
Rationale: The main function may require special actions.
Effect on original feature: Deletion of semantically well-defined feature

Difficulty of converting: Trivial: create an intermediary function such as `mymain(argc, argv)`.
How widely used: Seldom

3.2955
Change: C allows "compatible types" in several places, C++ does not For example, otherwise-identical `struct` types with different tag names are "compatible" in C but are distinctly different types in C++.
Rationale: Stricter type checking is essential for C++.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Semantic transformation. The "typesafe linkage" mechanism will find many, but not all, of such problems. Those problems not found by typesafe linkage will continue to function properly, according to the "layout compatibility rules" of this International Standard.
How widely used: Common.

C.1.3 Clause 4: standard conversions [diff.conv]

4.579
Change: Converting `void*` to a pointer-to-object type requires casting

```
char a[10];
void* b=a;
void foo() {
  char* c=b;
}
```

ISO C will accept this usage of pointer to void being assigned to a pointer to object type. C++ will not.
Rationale: C++ tries harder than C to enforce compile-time type safety.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Could be automated. Violations will be diagnosed by the C++ translator. The fix is to add a cast. For example:

```
char* c = (char*) b;
```

How widely used: This is fairly widely used but it is good programming practice to add the cast when assigning pointer-to-void to pointer-to-object. Some ISO C translators will give a warning if the cast is not used.

C.1.4 Clause 5: expressions [diff.expr]

5.955.1
Change: Implicit declaration of functions is not allowed
Rationale: The type-safe nature of C++.
Effect on original feature: Deletion of semantically well-defined feature. Note: the original feature was labeled as "obsolescent" in ISO C.
Difficulty of converting: Syntactic transformation. Facilities for producing explicit function declarations are fairly widespread commercially.
How widely used: Common.

5.2075.1, 5.2765
Change: Types must be declared in declarations, not in expressions In C, a sizeof expression or cast expression may create a new type. For example,

```
p = (void*)(struct x {int i;} *)0;
```

declares a new type, struct x .
Rationale: This prohibition helps to clarify the location of declarations in the source code.

Effect on original feature: Deletion of a semantically well-defined feature.
Difficulty of converting: Syntactic transformation.
How widely used: Seldom.

5.3337, 5.3483, 5.3602

Change: The result of a conditional expression, an assignment expression, or a comma expression may be an lvalue
Rationale: C++ is an object-oriented language, placing relatively more emphasis on lvalues. For example, functions may return lvalues.
Effect on original feature: Change to semantics of well-defined feature. Some C expressions that implicitly rely on lvalue-to-rvalue conversions will yield different results. For example,

```
char arr[100];
sizeof(0, arr)
```

yields `100` in C++ and `sizeof(char*)` in C.
Difficulty of converting: Programs must add explicit casts to the appropriate rvalue.
How widely used: Rare.

C.1.5 Clause 6: statements [diff.stat]

6.72.1, 6.413.1
Change: It is now invalid to jump past a declaration with explicit or implicit initializer (except across entire block not entered)
Rationale: Constructors used in initializers may allocate resources which need to be de-allocated upon leaving the block. Allowing jump past initializers would require complicated run-time determination of allocation. Furthermore, any use of the uninitialized object could be a disaster. With this simple compile-time rule, C++ assures that if an initialized variable is in scope, then it has assuredly been initialized.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Semantic transformation.
How widely used: Seldom.

6.270.2
Change: It is now invalid to return (explicitly or implicitly) from a function which is declared to return a value without actually returning a value
Rationale: The caller and callee may assume fairly elaborate return-value mechanisms for the return of class objects. If some flow paths execute a return without specifying any value, the implementation must embody many more complications. Besides, promising to return a value of a given type, and then not returning such a value, has always been recognized to be a questionable practice, tolerated only because very-old C had no distinction between void functions and int functions.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Semantic transformation. Add an appropriate return value to the source code, such as zero.
How widely used: Seldom. For several years, many existing C implementations have produced warnings in this case.

C.1.6 Clause 7: declarations [diff.dcl]

7.49.1
Change: In C++, the `static` or `extern` specifiers can only be applied to names of objects or functions Using these specifiers with type declarations is illegal in C++. In C, these specifiers are ignored when used on type declarations.

Example:

```
static struct S {                // valid C, invalid in C++
  int i;
};
```

Rationale: Storage class specifiers don't have any meaning when associated with a type. In C++, class members can be declared with the `static` storage class specifier. Allowing storage class specifiers on type declarations could render the code confusing for users.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Syntactic transformation.
How widely used: Seldom.

7.235.1
Change: A C++ typedef name must be different from any class type name declared in the same scope (except if the typedef is a synonym of the class name with the same name). In C, a typedef name and a struct tag name declared in the same scope can have the same name (because they have different name spaces)

Example:

```
typedef struct name1 { /*...*/ } name1;          // valid C and C++
struct name { /*...*/ };
typedef int name;                // valid C, invalid C++
```

Rationale: For ease of use, C++ doesn't require that a type name be prefixed with the keywords `class`, `struct` or `union` when used in object declarations or type casts.

Example:

```
class name { /*...*/ };
name i;                          // i has type class name
```

Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Semantic transformation. One of the 2 types has to be renamed.
How widely used: Seldom.

7.519.1 [see also 3.1459]
Change: const objects must be initialized in C++ but can be left uninitialized in C
Rationale: A const object cannot be assigned to so it must be initialized to hold a useful value.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Semantic transformation.
How widely used: Seldom.

7.519.1
Change: Banning implicit int

In C++ a *decl-specifier-seq* must contain a *type-specifier*, unless it is followed by a declarator for a constructor, a destructor, or a conversion function. In the following example, the left-hand column presents valid C; the right-hand column presents equivalent C++:

```
void f(const parm);              void f(const int parm);
const n = 3;                     const int n = 3;
main()                           int main()
    /* ... */                        /* ... */
```

Rationale: In C++, implicit int creates several opportunities for ambiguity between expressions involving function-like casts and declarations. Explicit declaration is increasingly considered to be proper style. Liaison with WG14 (C) indicated support for (at least) deprecating implicit int in the next revision of C.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Syntactic transformation. Could be automated.
How widely used: Common.

7.832.0.1
Change: The keyword `auto` cannot be used as a storage class specifier.

```
void f() {
  auto int x;      // valid C, invalid C++
}
```

Rationale: Allowing the use of `auto` to deduce the type of a variable from its initializer results in undesired interpretations of `auto` as a storage class specifier in certain contexts.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Syntactic transformation.
How widely used: Rare.

7.1001
Change: C++ objects of enumeration type can only be assigned values of the same enumeration type. In C, objects of enumeration type can be assigned values of any integral type

Example:

```
enum color { red, blue, green };
enum color c = 1;                // valid C, invalid C++
```

Rationale: The type-safe nature of C++.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Syntactic transformation. (The type error produced by the assignment can be automatically corrected by applying an explicit cast.)
How widely used: Common.

7.1001
Change: In C++, the type of an enumerator is its enumeration. In C, the type of an enumerator is `int`.

Example:

```
enum e { A };
sizeof(A) == sizeof(int)         // in C
sizeof(A) == sizeof(e)           // in C++
/* and sizeof(int) is not necessarily equal to sizeof(e) */
```

Rationale: In C++, an enumeration is a distinct type.
Effect on original feature: Change to semantics of well-defined feature.
Difficulty of converting: Semantic transformation.
How widely used: Seldom. The only time this affects existing C code is when the size of an enumerator is taken. Taking the size of an enumerator is not a common C coding practice.

C.1.7 Clause 8: declarators [diff.decl]

8.360.1

Change: In C++, a function declared with an empty parameter list takes no arguments. In C, an empty parameter list means that the number and type of the function arguments are unknown.

Example:

```
int f();            // means int f(void) in C++
                    // int f( unknown ) in C
```

Rationale: This is to avoid erroneous function calls (i.e., function calls with the wrong number or type of arguments).
Effect on original feature: Change to semantics of well-defined feature. This feature was marked as "obsolescent" in C.
Difficulty of converting: Syntactic transformation. The function declarations using C incomplete declaration style must be completed to become full prototype declarations. A program may need to be updated further if different calls to the same (non-prototype) function have different numbers of arguments or if the type of corresponding arguments differed.
How widely used: Common.

8.360.1 [see 5.2075.1]

Change: In C++, types may not be defined in return or parameter types. In C, these type definitions are allowed

Example:

```
void f( struct S { int a; } arg ) {}     // valid C, invalid C++
enum E { A, B, C } f() {}                // valid C, invalid C++
```

Rationale: When comparing types in different compilation units, C++ relies on name equivalence when C relies on structural equivalence. Regarding parameter types: since the type defined in an parameter list would be in the scope of the function, the only legal calls in C++ would be from within the function itself.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Semantic transformation. The type definitions must be moved to file scope, or in header files.
How widely used: Seldom. This style of type definitions is seen as poor coding style.

8.706

Change: In C++, the syntax for function definition excludes the "old-style" C function. In C, "old-style" syntax is allowed, but deprecated as "obsolescent."
Rationale: Prototypes are essential to type safety.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Syntactic transformation.
How widely used: Common in old programs, but already known to be obsolescent.

8.2115.1

Change: In C++, when initializing an array of character with a string, the number of characters in the string (including the terminating `'\0'`) must not exceed the number of elements in the array. In C, an array can be initialized with a string even if the array is not large enough to contain the string-terminating `'\0'`

Example:

```
char array[4] = "abcd";          // valid C, invalid C++
```

Rationale: When these non-terminated arrays are manipulated by standard string routines, there is potential for major catastrophe.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Semantic transformation. The arrays must be declared one element bigger to contain the string terminating `'\0'`.
How widely used: Seldom. This style of array initialization is seen as poor coding style.

C.1.8 Clause 9: classes [diff.class]

9.172 [see also 7.235.1]
Change: In C++, a class declaration introduces the class name into the scope where it is declared and hides any object, function or other declaration of that name in an enclosing scope. In C, an inner scope declaration of a struct tag name never hides the name of an object or function in an outer scope

Example:

```
int x[99];
void f() {
  struct x { int a; };
  sizeof(x);  /* size of the array in C */
  /* size of the struct in C++ */
}
```

Rationale: This is one of the few incompatibilities between C and C++ that can be attributed to the new C++ name space definition where a name can be declared as a type and as a non-type in a single scope causing the non-type name to hide the type name and requiring that the keywords `class`, `struct`, `union` or `enum` be used to refer to the type name. This new name space definition provides important notational conveniences to C++ programmers and helps making the use of the user-defined types as similar as possible to the use of fundamental types. The advantages of the new name space definition were judged to outweigh by far the incompatibility with C described above.
Effect on original feature: Change to semantics of well-defined feature.
Difficulty of converting: Semantic transformation. If the hidden name that needs to be accessed is at global scope, the `::` C++ operator can be used. If the hidden name is at block scope, either the type or the struct tag has to be renamed.
How widely used: Seldom.

9.921
Change: Bit-fields of type plain `int` are signed.
Rationale: Leaving the choice of signedness to implementations could lead to inconsistent definitions of template specializations. For consistency, the implementation freedom was eliminated for non-dependent types, too.
Effect on original feature: The choise is implementation-defined in C, but not so in C++.
Difficulty of converting: Syntactic transformation.
How widely used: Seldom.

9.960
Change: In C++, the name of a nested class is local to its enclosing class. In C the name of the nested class belongs to the same scope as the name of the outermost enclosing class.

Example:

```
struct X {
  struct Y { /* ... */ } y;
};
```

```
struct Y yy;                    // valid C, invalid C++
```

Rationale: C++ classes have member functions which require that classes establish scopes. The C rule would leave classes as an incomplete scope mechanism which would prevent C++ programmers from maintaining locality within a class. A coherent set of scope rules for C++ based on the C rule would be very complicated and C++ programmers would be unable to predict reliably the meanings of nontrivial examples involving nested or local functions.
Effect on original feature: Change of semantics of well-defined feature.
Difficulty of converting: Semantic transformation. To make the struct type name visible in the scope of the enclosing struct, the struct tag could be declared in the scope of the enclosing struct, before the enclosing struct is defined. Example:

```
struct Y;                       // struct Y and struct X are at the same scope
struct X {
  struct Y { /* ... */ } y;
};
```

All the definitions of C struct types enclosed in other struct definitions and accessed outside the scope of the enclosing struct could be exported to the scope of the enclosing struct. Note: this is a consequence of the difference in scope rules, which is documented in 3.464.
How widely used: Seldom.

9.992
Change: In C++, a typedef name may not be redeclared in a class definition after being used in that definition

Example:

```
typedef int I;
struct S {
  I i;
  int I;                    // valid C, invalid C++
};
```

Rationale: When classes become complicated, allowing such a redefinition after the type has been used can create confusion for C++ programmers as to what the meaning of 'I' really is.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Semantic transformation. Either the type or the struct member has to be renamed.
How widely used: Seldom.

C.1.9 Clause 12: special member functions [diff.special]

12.1577
Change: Copying volatile objects

The implicitly-declared copy constructor and implicitly-declared copy assignment operator cannot make a copy of a volatile lvalue. For example, the following is valid in ISO C:

```
struct X { int i; };
volatile struct X x1 = {0};
struct X x2(x1);                // invalid C++
struct X x3;
x3 = x1;                        // also invalid C++
```

Rationale: Several alternatives were debated at length. Changing the parameter to `volatile const X&` would greatly complicate the generation of efficient code for class objects. Discussion of providing two alternative signatures for these implicitly-defined operations raised unanswered concerns about creating ambiguities and complicating the rules that specify the formation of these operators according to the bases and members.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Semantic transformation. If volatile semantics are required for the copy, a user-declared constructor or assignment must be provided. [*Note:* This user-declared constructor may be explicitly defaulted. — *end note*] If non-volatile semantics are required, an explicit `const_cast` can be used.
How widely used: Seldom.

C.1.10 Clause 16: preprocessing directives [diff.cpp]

16.238
Change: Whether `__STDC__` is defined and if so, what its value is, are implementation-defined
Rationale: C++ is not identical to ISO C. Mandating that `__STDC__` be defined would require that translators make an incorrect claim. Each implementation must choose the behavior that will be most useful to its marketplace.
Effect on original feature: Change to semantics of well-defined feature.
Difficulty of converting: Semantic transformation.
How widely used: Programs and headers that reference `__STDC__` are quite common.

C.2 C++ and ISO C++ 2003 [diff.cpp03]

1 This subclause lists the differences between C++ and ISO C++ 2003 (ISO/IEC 14882:2003, *Programming Languages — C++*), by the chapters of this document.

C.2.1 Clause 2: lexical conventions [diff.cpp03.lex]

2.214
Change: New kinds of string literals
Rationale: Required for new features.
Effect on original feature: Valid C++ 2003 code may fail to compile or produce different results in this International Standard. Specifically, macros named `R`, `u8`, `u8R`, `u`, `uR`, `U`, `UR`, or `LR` will not be expanded when adjacent to a string literal but will be interpreted as part of the string literal. For example,

```
#define u8 "abc"
const char* s = u8"def";        // Previously "abcdef", now "def"
```

2.214
Change: User-defined literal string support
Rationale: Required for new features.
Effect on original feature: Valid C++ 2003 code may fail to compile or produce different results in this International Standard, as the following example illustrates.

```
#define _x "there"
"hello"_x         // #1
```

Previously, #1 would have consisted of two separate preprocessing tokens and the macro `_x` would have been expanded. In this International Standard, #1 consists of a single preprocessing tokens, so the macro is not expanded.

2.287
Change: New keywords

Rationale: Required for new features.
Effect on original feature: Added to Table 4, the following identifiers are new keywords: `alignas`, `alignof`, `char16_t`, `char32_t`, `constexpr`, `decltype`, `noexcept`, `nullptr`, `static_assert`, and `thread_local`. Valid C++ 2003 code using these identifiers is invalid in this International Standard.

2.300.1
Change: Type of integer literals
Rationale: C99 compatibility.
Effect on original feature: Certain integer literals larger than can be represented by `long` could change from an unsigned integer type to `signed long long`.

C.2.2 Clause 4: standard conversions [diff.cpp03.conv]

4.579
Change: Only literals are integer null pointer constants
Rationale: Removing surprising interactions with templates and constant expressions
Effect on original feature: Valid C++ 2003 code may fail to compile or produce different results in this International Standard, as the following example illustrates:

```
void f(void *);  // #1
void f(...);     // #2
template<int N> void g() {
  f(0*N);        // calls #2; used to call #1
}
```

C.2.3 Clause 5: expressions [diff.cpp03.expr]

5.2851
Change: Specify rounding for results of integer `/` and `%`
Rationale: Increase portability, C99 compatibility.
Effect on original feature: Valid C++ 2003 code that uses integer division rounds the result toward 0 or toward negative infinity, whereas this International Standard always rounds the result toward 0.

C.2.4 Clause 7: declarations [diff.cpp03.dcl.dcl]

7.40
Change: Remove `auto` as a storage class specifier
Rationale: New feature.
Effect on original feature: Valid C++ 2003 code that uses the keyword `auto` as a storage class specifier may be invalid in this International Standard. In this International Standard, `auto` indicates that the type of a variable is to be deduced from its initializer expression.

C.2.5 Clause 8: declarators [diff.cpp03.dcl.decl]

8.2435.1
Change: Narrowing restrictions in aggregate initializers
Rationale: Catches bugs.
Effect on original feature: Valid C++ 2003 code may fail to compile in this International Standard. For example, the following code is valid in C++ 2003 but invalid in this International Standard because `double` to `int` is a narrowing conversion:

```
int x[] = { 2.0 };
```

C.2.6 Clause 12: special member functions [diff.cpp03.special]

12.30, 12.807, 12.1577
Change: Implicitly-declared special member functions are defined as deleted when the implicit definition

would have been ill-formed.
Rationale: Improves template argument deduction failure.
Effect on original feature: A valid C++ 2003 program that uses one of these special member functions in a context where the definition is not required (e.g., in an expression that is not potentially evaluated) becomes ill-formed.

12.807 (destructors)
Change: User-declared destructors have an implicit exception specification.
Rationale: Clarification of destructor requirements.
Effect on original feature: Valid C++ 2003 code may execute differently in this International Standard. In particular, destructors that throw exceptions will call `std::terminate()` (without calling `std::unexpected()`) if their exception specification is `noexcept` or `noexcept(true)`. For a throwing virtual destructor of a derived class, `std::terminate()` can be avoided only if the base class virtual destructor has an exception specification that is not `noexcept` and not `noexcept(true)`.

C.2.7 Clause 14: templates [diff.cpp03.temp]

14.97
Change: Remove `export`
Rationale: No implementation consensus.
Effect on original feature: A valid C++ 2003 declaration containing `export` is ill-formed in this International Standard.

14.326
Change: Remove whitespace requirement for nested closing template right angle brackets
Rationale: Considered a persistent but minor annoyance. Template aliases representing nonclass types would exacerbate whitespace issues.
Effect on original feature: Change to semantics of well-defined expression. A valid C++ 2003 expression containing a right angle bracket ("`>`") followed immediately by another right angle bracket may now be treated as closing two templates. For example, the following code is valid in C++ 2003 because "`>>`" is a right-shift operator, but invalid in this International Standard because "`>>`" closes two templates.

```
template <class T> struct X { };
template <int N> struct Y { };
X< Y< 1 >> 2 > > x;
```

14.1532.0.1
Change: Allow dependent calls of functions with internal linkage
Rationale: Overly constrained, simplify overload resolution rules.
Effect on original feature: A valid C++ 2003 program could get a different result than this International Standard.

C.2.8 Clause 17: library introduction [diff.cpp03.library]

17 – 30
Change: New reserved identifiers
Rationale: Required by new features.
Effect on original feature: Valid C++ 2003 code that uses any identifiers added to the C++ standard library by this International Standard may fail to compile or produce different results in This International Standard. A comprehensive list of identifiers used by the C++ standard library can be found in the Index of Library Names in this International Standard.

17.48.0.1
Change: New headers
Rationale: New functionality.

Effect on original feature: The following C++ headers are new: `<array>`, `<atomic>`, `<chrono>`, `<codecvt>`, `<condition_variable>`, `<forward_list>`, `<future>`, `<initializer_list>`, `<mutex>`, `<random>`, `<ratio>`, `<regex>`, `<scoped_allocator>`, `<system_error>`, `<thread>`, `<tuple>`, `<typeindex>`, `<type_traits>`, `<unordered_map>`, and `<unordered_set>`. In addition the following C compatibility headers are new: `<ccomplex>`, `<cfenv>`, `<cinttypes>`, `<cstdalign>`, `<cstdbool>`, `<cstdint>`, `<ctgmath>`, and `<cuchar>`. Valid C++ 2003 code that `#include`s headers with these names may be invalid in this International Standard.

17.143.0.1
Effect on original feature: Function `swap` moved to a different header
Rationale: Remove dependency on `<algorithm>` for `swap`.
Effect on original feature: Valid C++ 2003 code that has been compiled expecting swap to be in `<algorithm>` may have to instead include `<utility>`.

17.295.0.0.1
Change: New reserved namespace
Rationale: New functionality.
Effect on original feature: The global namespace `posix` is now reserved for standardization. Valid C++ 2003 code that uses a top-level namespace `posix` may be invalid in this International Standard.

17.563.0.1
Change: Additional restrictions on macro names
Rationale: Avoid hard to diagnose or non-portable constructs.
Effect on original feature: Names of attribute identifiers may not be used as macro names. Valid C++ 2003 code that defines `override`, `final`, `carries_dependency`, or `noreturn` as macros is invalid in this International Standard.

C.2.9 Clause 18: language support library [diff.cpp03.language.support]

18.159.1.1
Change: Linking `new` and `delete` operators
Rationale: The two throwing single-object signatures of `operator new` and `operator delete` are now specified to form the base functionality for the other operators. This clarifies that replacing just these two signatures changes others, even if they are not explicitly changed.
Effect on original feature: Valid C++ 2003 code that replaces global `new` or `delete` operators may execute differently in this International Standard. For example, the following program should write `"custom deallocation"` twice, once for the single-object delete and once for the array delete.

```
#include <cstdio>
#include <cstdlib>
#include <new>

void* operator new(std::size_t size) throw(std::bad_alloc) {
  return std::malloc(size);
}

void operator delete(void* ptr) throw() {
  std::puts("custom deallocation");
  std::free(ptr);
}

int main() {
  int* i = new int;
  delete i;                    // single-object delete
  int* a = new int[3];
  delete [] a;                 // array delete
```

```
  return 0;
}
```

18.159.1.1
Change: `operator new` may throw exceptions other than `std::bad_alloc`
Rationale: Consistent application of `noexcept`.
Effect on original feature: Valid C++ 2003 code that assumes that global `operator new` only throws `std::bad_alloc` may execute differently in this International Standard.

C.2.10 Clause 19: diagnostics library [diff.cpp03.diagnostics]

19.12
Change: Thread-local error numbers
Rationale: Support for new thread facilities.
Effect on original feature: Valid but implementation-specific C++ 2003 code that relies on `errno` being the same across threads may change behavior in this International Standard.

C.2.11 Clause 20: general utilities library [diff.cpp03.utilities]

20.85.1
Change: Minimal support for garbage-collected regions
Rationale: Required by new feature.
Effect on original feature: Valid C++ 2003 code, compiled without traceable pointer support, that interacts with newer C++ code using regions declared reachable may have different runtime behavior.

20.339.1, 20.339.2, 20.339.3, 20.357.1, 20.357.2, 20.357.3
Change: Standard function object types no longer derived from `std::unary_function` or `std::binary_function`
Rationale: Superseded by new feature.
Effect on original feature: Valid C++ 2003 code that depends on function object types being derived from `unary_function` or `binary_function` will execute differently in this International Standard.

C.2.12 Clause 21: strings library [diff.cpp03.strings]

21.26
Change: `basic_string` requirements no longer allow reference-counted strings
Rationale: Invalidation is subtly different with reference-counted strings. This change regularizes behavior for this International Standard.
Effect on original feature: Valid C++ 2003 code may execute differently in this International Standard.

21.31.1
Change: Loosen `basic_string` invalidation rules
Rationale: Allow small-string optimization.
Effect on original feature: Valid C++ 2003 code may execute differently in this International Standard. Some `const` member functions, such as `data` and `c_str`, no longer invalidate iterators.

C.2.13 Clause 23: containers library [diff.cpp03.containers]

23.2
Change: Complexity of `size()` member functions now constant
Rationale: Lack of specification of complexity of `size()` resulted in divergent implementations with inconsistent performance characteristics.
Effect on original feature: Some container implementations that conform to C++ 2003 may not conform to the specified `size()` requirements in this International Standard. Adjusting containers such as `std::list` to the stricter requirements may require incompatible changes.

23.2
Change: Requirements change: relaxation
Rationale: Clarification.
Effect on original feature: Valid C++ 2003 code that attempts to meet the specified container requirements may now be over-specified. Code that attempted to be portable across containers may need to be adjusted as follows:

— not all containers provide `size()`; use `empty()` instead of `size() == 0`;

— not all containers are empty after construction (`array`);

— not all containers have constant complexity for `swap()` (`array`).

23.2
Change: Requirements change: default constructible
Rationale: Clarification of container requirements.
Effect on original feature: Valid C++ 2003 code that attempts to explicitly instantiate a container using a user-defined type with no default constructor may fail to compile.

23.224.1, 23.294.1
Change: Signature changes: from `void` return types
Rationale: Old signature threw away useful information that may be expensive to recalculate.
Effect on original feature: The following member functions have changed:

— `erase(iter)` for `set`, `multiset`, `map`, `multimap`

— `erase(begin, end)` for `set`, `multiset`, `map`, `multimap`

— `insert(pos, num, val)` for `vector`, `deque`, `list`, `forward_list`

— `insert(pos, beg, end)` for `vector`, `deque`, `list`, `forward_list`

Valid C++ 2003 code that relies on these functions returning `void` (e.g., code that creates a pointer to member function that points to one of these functions) will fail to compile with this International Standard.

23.224.1, 23.294.1
Change: Signature changes: from `iterator` to `const_iterator` parameters
Rationale: Overspecification. *Effects:* The signatures of the following member functions changed from taking an `iterator` to taking a `const_iterator`:

— `insert(iter, val)` for `vector`, `deque`, `list`, `set`, `multiset`, `map`, `multimap`

— `insert(pos, beg, end)` for `vector`, `deque`, `list`, `forward_list`

— `erase(iter)` for `set`, `multiset`, `map`, `multimap`

— `erase(begin, end)` for `set`, `multiset`, `map`, `multimap`

— all forms of `list::splice`

— all forms of `list::merge`

Valid C++ 2003 code that uses these functions may fail to compile with this International Standard.

23.224.1, 23.294.1
Change: Signature changes: `resize`
Rationale: Performance, compatibility with move semantics.
Effect on original feature: For `vector`, `deque`, and `list` the fill value passed to `resize` is now passed by reference instead of by value, and an additional overload of `resize` has been added. Valid C++ 2003 code that uses this function may fail to compile with this International Standard.

C.2.14 Clause 25: algorithms library [diff.cpp03.algorithms]

25.1

Change: Result state of inputs after application of some algorithms
Rationale: Required by new feature.
Effect on original feature: A valid C++ 2003 program may detect that an object with a valid but unspecified state has a different valid but unspecified state with this International Standard. For example, `std::remove` and `std::remove_if` may leave the tail of the input sequence with a different set of values than previously.

C.2.15 Clause 26: numerics library [diff.cpp03.numerics]

26.11

Change: Specified representation of complex numbers
Rationale: Compatibility with C99.
Effect on original feature: Valid C++ 2003 code that uses implementation-specific knowledge about the binary representation of the required template specializations of `std::complex` may not be compatible with this International Standard.

C.2.16 Clause 27: Input/output library [diff.cpp03.input.output]

27.117.0.0.2, 27.190.0.2, 27.84.0.1

Change: Specify use of explicit in existing boolean conversion operators
Rationale: Clarify intentions, avoid workarounds.
Effect on original feature: Valid C++ 2003 code that relies on implicit boolean conversions will fail to compile with this International Standard. Such conversions occur in the following conditions:

— passing a value to a function that takes an argument of type `bool`;

— using `operator==` to compare to `false` or `true`;

— returning a value from a function with a return type of `bool`;

— initializing members of type `bool` via aggregate initialization;

— initializing a `const bool&` which would bind to a temporary.

27.59.0.1.1

Change: Change base class of `std::ios_base::failure`
Rationale: More detailed error messages.
Effect on original feature: `std::ios_base::failure` is no longer derived directly from `std::exception`, but is now derived from `std::system_error`, which in turn is derived from `std::runtime_error`. Valid C++ 2003 code that assumes that `std::ios_base::failure` is derived directly from `std::exception` may execute differently in this International Standard.

27.58.1

Change: Flag types in `std::ios_base` are now bitmasks with values defined as constexpr static members
Rationale: Required for new features.
Effect on original feature: Valid C++ 2003 code that relies on `std::ios_base` flag types being represented as `std::bitset` or as an integer type may fail to compile with this International Standard. For example:

```
#include <iostream>

int main() {
  int flag = std::ios_base::hex;
  std::cout.setf(flag);          // error: setf does not take argument of type int
  return 0;
}
```

C.3 C++ and ISO C++ 2011 [diff.cpp11]

1 This subclause lists the differences between C++ and ISO C++ 2011 (ISO/IEC 14882:2011, *Programming Languages — C++*), by the chapters of this document.

C.3.1 Clause 2: lexical conventions [diff.cpp11.lex]

2.266
Change: *pp-number* can contain one or more single quotes.
Rationale: Necessary to enable single quotes as digit separators.
Effect on original feature: Valid C++ 2011 code may fail to compile or may change meaning in this International Standard. For example, the following code is valid both in C++ 2011 and in this International Standard, but the macro invocation produces different outcomes because the single quotes delimit a character literal in C++ 2011, whereas they are digit separators in this International Standard:

```
#define M(x, ...) __VA_ARGS__
int x[2] = { M(1'2,3'4) };
// int x[2] = {};       — C++ 2011
// int x[2] = { 3'4 };  — this International Standard
```

C.3.2 Clause 3: basic concepts [diff.cpp11.basic]

3.2428.0.1
Change: New usual (non-placement) deallocator
Rationale: Required for sized deallocation.
Effect on original feature: Valid C++ 2011 code could declare a global placement allocation function and deallocation function as follows:

```
void operator new(std::size_t, std::size_t);
void operator delete(void*, std::size_t) noexcept;
```

In this International Standard, however, the declaration of `operator delete` might match a predefined usual (non-placement) `operator delete` (3.2331.1). If so, the program is ill-formed, as it was for class member allocation functions and deallocation functions (5.2221.1).

C.3.3 Clause 7: declarations [diff.cpp11.dcl.dcl]

7.328.1
Change: `constexpr` non-static member functions are not implicitly `const` member functions.
Rationale: Necessary to allow `constexpr` member functions to mutate the object.
Effect on original feature: Valid C++ 2011 code may fail to compile in this International Standard. For example, the following code is valid in C++ 2011 but invalid in this International Standard because it declares the same member function twice with different return types:

```
struct S {
  constexpr const int &f();
  int &f();
};
```

C.3.4 Clause 27: input/output library [diff.cpp11.input.output]

27.310.1
Change: `gets` is not defined.
Rationale: Use of `gets` is considered dangerous.
Effect on original feature: Valid C++ 2011 code that uses the `gets` function may fail to compile in this International Standard.

C.4 C standard library [diff.library]

1

This subclause summarizes the contents of the C++ standard library included from the Standard C library. It also summarizes the explicit changes in definitions, declarations, or behavior from the Standard C library noted in other subclauses (17.48.0.1, 18.2, 21.261).

1 The C++ standard library provides 57 standard macros from the C library, as shown in Table 150.

2 The header names (enclosed in < and >) indicate that the macro may be defined in more than one header. All such definitions are equivalent (3.91).

Table 150 — Standard macros

assert	HUGE_VAL	NULL <cstring>	SIGINT	va_end
BUFSIZ	LC_ALL	NULL <ctime>	SIGSEGV	va_start
CLOCKS_PER_SEC	LC_COLLATE	NULL <cwchar>	SIGTERM	WCHAR_MAX
EDOM	LC_CTYPE	offsetof	SIG_DFL	WCHAR_MIN
EILSEQ	LC_MONETARY	RAND_MAX	SIG_ERR	WEOF <cwchar>
EOF	LC_NUMERIC	SEEK_CUR	SIG_IGN	WEOF <cwctype>
ERANGE	LC_TIME	SEEK_END	stderr	_IOFBF
errno	L_tmpnam	SEEK_SET	stdin	_IOLBF
EXIT_FAILURE	MB_CUR_MAX	setjmp	stdout	_IONBF
EXIT_SUCCESS	NULL <clocale>	SIGABRT	TMP_MAX	
FILENAME_MAX	NULL <cstddef>	SIGFPE	va_arg	
FOPEN_MAX	NULL <cstdlib>	SIGILL	va_copy	

3 The C++ standard library provides 57 standard values from the C library, as shown in Table 151.

Table 151 — Standard values

CHAR_BIT	FLT_DIG	INT_MIN	MB_LEN_MAX
CHAR_MAX	FLT_EPSILON	LDBL_DIG	SCHAR_MAX
CHAR_MIN	FLT_MANT_DIG	LDBL_EPSILON	SCHAR_MIN
DBL_DIG	FLT_MAX	LDBL_MANT_DIG	SHRT_MAX
DBL_EPSILON	FLT_MAX_10_EXP	LDBL_MAX	SHRT_MIN
DBL_MANT_DIG	FLT_MAX_EXP	LDBL_MAX_10_EXP	UCHAR_MAX
DBL_MAX	FLT_MIN	LDBL_MAX_EXP	UINT_MAX
DBL_MAX_10_EXP	FLT_MIN_10_EXP	LDBL_MIN	ULONG_MAX
DBL_MAX_EXP	FLT_MIN_EXP	LDBL_MIN_10_EXP	USHRT_MAX
DBL_MIN	FLT_RADIX	LDBL_MIN_EXP	
DBL_MIN_10_EXP	FLT_ROUNDS	LONG_MAX	
DBL_MIN_EXP	INT_MAX	LONG_MIN	

4 The C++ standard library provides 20 standard types from the C library, as shown in Table 152.

Table 152 — Standard types

clock_t	ldiv_t	size_t <cstdio>	va_list
div_t	mbstate_t	size_t <cstdlib>	wctrans_t
FILE	ptrdiff_t	size_t <cstring>	wctype_t
fpos_t	sig_atomic_t	size_t <ctime>	wint_t <cwchar>
jmp_buf	size_t <cstddef>	time_t	wint_t <cwctype>

5 The C++ standard library provides 2 standard `struct`s from the C library, as shown in Table 153.

Table 153 — Standard structs

`lconv`	`tm`

6

The C++ standard library provides 209 standard functions from the C library, as shown in Table 154.

Table 154 — Standard functions

abort	fmod	iswalnum	modf	strlen	wcscat
abs	fopen	iswalpha	perror	strncat	wcschr
acos	fprintf	iswcntrl	pow	strncmp	wcscmp
asctime	fputc	iswctype	printf	strncpy	wcscoll
asin	fputs	iswdigit	putc	strpbrk	wcscpy
atan	fputwc	iswgraph	putchar	strrchr	wcscspn
atan2	fputws	iswlower	puts	strspn	wcsftime
atexit	fread	iswprint	putwc	strstr	wcslen
atof	free	iswpunct	putwchar	strtod	wcsncat
atoi	freopen	iswspace	qsort	strtok	wcsncmp
atol	frexp	iswupper	raise	strtol	wcsncpy
bsearch	fscanf	iswxdigit	rand	strtoul	wcspbrk
btowc	fseek	isxdigit	realloc	strxfrm	wcsrchr
calloc	fsetpos	labs	remove	swprintf	wcsrtombs
ceil	ftell	ldexp	rename	swscanf	wcsspn
clearerr	fwide	ldiv	rewind	system	wcsstr
clock	fwprintf	localeconv	scanf	tan	wcstod
cos	fwrite	localtime	setbuf	tanh	wcstok
cosh	fwscanf	log	setlocale	time	wcstol
ctime	getc	log10	setvbuf	tmpfile	wcstombs
difftime	getchar	longjmp	signal	tmpnam	wcstoul
div	getenv	malloc	sin	tolower	wcsxfrm
exit	getwc	mblen	sinh	toupper	wctob
exp	getwchar	mbrlen	sprintf	towctrans	wctomb
fabs	gmtime	mbrtowc	sqrt	towlower	wctrans
fclose	isalnum	mbsinit	srand	towupper	wctype
feof	isalpha	mbsrtowcs	sscanf	ungetc	wmemchr
ferror	iscntrl	mbstowcs	strcat	ungetwc	wmemcmp
fflush	isdigit	mbtowc	strchr	vfprintf	wmemcpy
fgetc	isgraph	memchr	strcmp	vfwprintf	wmemmove
fgetpos	islower	memcmp	strcoll	vprintf	wmemset
fgets	isprint	memcpy	strcpy	vsprintf	wprintf
fgetwc	ispunct	memmove	strcspn	vswprintf	wscanf
fgetws	isspace	memset	strerror	vwprintf	
floor	isupper	mktime	strftime	wcrtomb	

C.6.1 Modifications to headers [diff.mods.to.headers]

1

For compatibility with the Standard C library, the C++ standard library provides the C headers enumerated in D.9, but their use is deprecated in C++.

C.11.1 Modifications to definitions [diff.mods.to.definitions]

C.11.1.1 Types `char16_t` and `char32_t` [diff.char16]

1 The types `char16_t` and `char32_t` are distinct types rather than typedefs to existing integral types.

C.11.1.2 Type `wchar_t` [diff.wchar.t]

1 `wchar_t` is a keyword in this International Standard (2.287). It does not appear as a type name defined in any of `<cstddef>`, `<cstdlib>`, or `<cwchar>` (21.261).

C.11.1.3 Header `<iso646.h>` [diff.header.iso646.h]

1 The tokens `and`, `and_eq`, `bitand`, `bitor`, `compl`, `not_eq`, `not`, `or`, `or_eq`, `xor`, and `xor_eq` are keywords in this International Standard (2.287). They do not appear as macro names defined in `<ciso646>`.

C.11.1.4 Macro `NULL` [diff.null]

1

The macro `NULL`, defined in any of `<clocale>`, `<cstddef>`, `<cstdio>`, `<cstdlib>`, `<cstring>`, `<ctime>`, or `<cwchar>`, is an implementation-defined C++ null pointer constant in this International Standard (18.2).

C.32.1 Modifications to declarations [diff.mods.to.declarations]

1 Header `<cstring>`: The following functions have different declarations:

(1.1) — `strchr`

(1.2) — `strpbrk`

(1.3) — `strrchr`

(1.4) — `strstr`

(1.5) — `memchr`

21.261 describes the changes.

C.32.2 Modifications to behavior [diff.mods.to.behavior]

1 Header `<cstdlib>`: The following functions have different behavior:

(1.1) — `atexit`

(1.2) — `exit`

(1.3) — `abort`

18.127 describes the changes.

2 Header `<csetjmp>`: The following functions have different behavior:

(2.1) — `longjmp`

18.295 describes the changes.

C.32.2.1 Macro `offsetof(type, member-designator)` [diff.offsetof]

1

The macro `offsetof`, defined in `<cstddef>`, accepts a restricted set of `type` arguments in this International Standard. 18.2 describes the change.

C.33.0.1 Memory allocation functions [diff.malloc]

1 The functions `calloc`, `malloc`, and `realloc` are restricted in this International Standard. 20.116.1 describes the changes.

Annex D (normative) Compatibility features [depr]

1 This Clause describes features of the C++ Standard that are specified for compatibility with existing implementations.

2

These are deprecated features, where *deprecated* is defined as: Normative for the current edition of the Standard, but having been identified as a candidate for removal from future revisions. An implementation may declare library names and entities described in this section with the `deprecated` attribute (7.1576.1).

D.3 Increment operator with `bool` operand [depr.incr.bool]

1 The use of an operand of type `bool` with the `++` operator is deprecated (see 5.2054.1 and 5.1264.1).

D.4 `register` keyword [depr.register]

1

The use of the `register` keyword as a *storage-class-specifier* (7.49.1) is deprecated.

D.6 Implicit declaration of copy functions [depr.impldec]

1

The implicit definition of a copy constructor as defaulted is deprecated if the class has a user-declared copy assignment operator or a user-declared destructor. The implicit definition of a copy assignment operator as defaulted is deprecated if the class has a user-declared copy constructor or a user-declared destructor (12.807, 12.1577). In a future revision of this International Standard, these implicit definitions could become deleted (8.706).

D.8 Dynamic exception specifications [depr.except.spec]

1 The use of *dynamic-exception-specification*s is deprecated.

D.9 C standard library headers [depr.c.headers]

1

For compatibility with the C standard library and the C Unicode TR, the C++ standard library provides the 26 *C headers*, as shown in Table 155.

Table 155 — C headers

`<assert.h>`	`<inttypes.h>`	`<signal.h>`	`<stdio.h>`	`<wchar.h>`
`<complex.h>`	`<iso646.h>`	`<stdalign.h>`	`<stdlib.h>`	`<wctype.h>`
`<ctype.h>`	`<limits.h>`	`<stdarg.h>`	`<string.h>`	
`<errno.h>`	`<locale.h>`	`<stdbool.h>`	`<tgmath.h>`	
`<fenv.h>`	`<math.h>`	`<stddef.h>`	`<time.h>`	
`<float.h>`	`<setjmp.h>`	`<stdint.h>`	`<uchar.h>`	

1

Every C header, each of which has a name of the form `name.h`, behaves as if each name placed in the standard library namespace by the corresponding `c`*`name`* header is placed within the global namespace scope. It is unspecified whether these names are first declared or defined within namespace scope (3.736.1) of the namespace `std` and are then injected into the global namespace scope by explicit *using-declarations* (7.1311.1).

1

[*Example:* The header `<cstdlib>` assuredly provides its declarations and definitions within the namespace `std`. It may also provide these names within the global namespace. The header `<stdlib.h>` assuredly provides the same declarations and definitions within the global namespace, much as in the C Standard. It may also provide these names within the namespace `std`. — *end example*]

D.53 Old iostreams members [depr.ios.members]

1 The following member names are in addition to names specified in Clause 27:

```
namespace std {
  class ios_base {
  public:
    typedef T1 io_state;
    typedef T2 open_mode;
    typedef T3 seek_dir;
    typedef implementation-defined streamoff;
    typedef implementation-defined streampos;
    // remainder unchanged
  };
}
```

2 The type `io_state` is a synonym for an integer type (indicated here as `T1`) that permits certain member functions to overload others on parameters of type `iostate` and provide the same behavior.

3 The type `open_mode` is a synonym for an integer type (indicated here as `T2`) that permits certain member functions to overload others on parameters of type `openmode` and provide the same behavior.

4 The type `seek_dir` is a synonym for an integer type (indicated here as `T3`) that permits certain member functions to overload others on parameters of type `seekdir` and provide the same behavior.

5 The type `streamoff` is an implementation-defined type that satisfies the requirements of off_type in 27.2.2.

6 The type `streampos` is an implementation-defined type that satisfies the requirements of pos_type in 27.2.2.

7 An implementation may provide the following additional member function, which has the effect of calling `sbumpc()` (27.96.0.2.3):

```
namespace std {
  template<class charT, class traits = char_traits<charT> >
  class basic_streambuf {
  public:
    void stossc();
    // remainder unchanged
  };
}
```

8 An implementation may provide the following member functions that overload signatures specified in Clause 27:

```
namespace std {
  template<class charT, class traits> class basic_ios {
  public:
    void clear(io_state state);
```

```
    void setstate(io_state state);
    void exceptions(io_state);
    // remainder unchanged
  };

  class ios_base {
  public:
    // remainder unchanged
  };

  template<class charT, class traits = char_traits<charT> >
  class basic_streambuf {
  public:
    pos_type pubseekoff(off_type off, ios_base::seek_dir way,
              ios_base::open_mode which = ios_base::in | ios_base::out);
    pos_type pubseekpos(pos_type sp,
              ios_base::open_mode which);
    // remainder unchanged
  };

  template <class charT, class traits = char_traits<charT> >
  class basic_filebuf : public basic_streambuf<charT,traits> {
  public:
    basic_filebuf<charT,traits>* open
    (const char* s, ios_base::open_mode mode);
    // remainder unchanged
  };

  template <class charT, class traits = char_traits<charT> >
  class basic_ifstream : public basic_istream<charT,traits> {
  public:
    void open(const char* s, ios_base::open_mode mode);
    // remainder unchanged
  };

  template <class charT, class traits = char_traits<charT> >
  class basic_ofstream : public basic_ostream<charT,traits> {
  public:
    void open(const char* s, ios_base::open_mode mode);
    // remainder unchanged
  };
}
```

9 The effects of these functions is to call the corresponding member function specified in Clause 27.

D.54 `char*` streams [depr.str.strstreams]

1 The header `<strstream>` defines three types that associate stream buffers with character array objects and assist reading and writing such objects.

D.54.1 Class `strstreambuf` [depr.strstreambuf]

```
namespace std {
  class strstreambuf : public basic_streambuf<char> {
  public:
    explicit strstreambuf(streamsize alsize_arg = 0);
```

```
    strstreambuf(void* (*palloc_arg)(size_t), void (*pfree_arg)(void*));
    strstreambuf(char* gnext_arg, streamsize n, char* pbeg_arg = 0);
    strstreambuf(const char* gnext_arg, streamsize n);

    strstreambuf(signed char* gnext_arg, streamsize n,
                 signed char* pbeg_arg = 0);
    strstreambuf(const signed char* gnext_arg, streamsize n);
    strstreambuf(unsigned char* gnext_arg, streamsize n,
                 unsigned char* pbeg_arg = 0);
    strstreambuf(const unsigned char* gnext_arg, streamsize n);

    virtual ~strstreambuf();

    void  freeze(bool freezefl = true);
    char* str();
    int   pcount();

  protected:
    virtual int_type overflow (int_type c = EOF);
    virtual int_type pbackfail(int_type c = EOF);
    virtual int_type underflow();
    virtual pos_type seekoff(off_type off, ios_base::seekdir way,
                             ios_base::openmode which
                               = ios_base::in | ios_base::out);
    virtual pos_type seekpos(pos_type sp, ios_base::openmode which
                               = ios_base::in | ios_base::out);
    virtual streambuf* setbuf(char* s, streamsize n);

  private:
    typedef T1 strstate;              // exposition only
    static const strstate allocated;  // exposition only
    static const strstate constant;   // exposition only
    static const strstate dynamic;    // exposition only
    static const strstate frozen;     // exposition only
    strstate strmode;                 // exposition only
    streamsize alsize;                // exposition only
    void* (*palloc)(size_t);          // exposition only
    void (*pfree)(void*);             // exposition only
  };
}
```

1 The class `strstreambuf` associates the input sequence, and possibly the output sequence, with an object of some *character* array type, whose elements store arbitrary values. The array object has several attributes.

2 [*Note:* For the sake of exposition, these are represented as elements of a bitmask type (indicated here as `T1`) called `strstate`. The elements are:

(2.1) — `allocated`, set when a dynamic array object has been allocated, and hence should be freed by the destructor for the `strstreambuf` object;

(2.2) — `constant`, set when the array object has `const` elements, so the output sequence cannot be written;

(2.3) — `dynamic`, set when the array object is allocated (or reallocated) as necessary to hold a character sequence that can change in length;

(2.4) — `frozen`, set when the program has requested that the array object not be altered, reallocated, or freed.

— *end note*]

3 [*Note:* For the sake of exposition, the maintained data is presented here as:

(3.1) — `strstate strmode`, the attributes of the array object associated with the `strstreambuf` object;

(3.2) — `int alsize`, the suggested minimum size for a dynamic array object;

(3.3) — `void* (*palloc)(size_t)`, points to the function to call to allocate a dynamic array object;

(3.4) — `void (*pfree)(void*)`, points to the function to call to free a dynamic array object.

— *end note*]

4 Each object of class `strstreambuf` has a *seekable area*, delimited by the pointers `seeklow` and `seekhigh`. If `gnext` is a null pointer, the seekable area is undefined. Otherwise, `seeklow` equals `gbeg` and `seekhigh` is either `pend`, if `pend` is not a null pointer, or `gend`.

D.54.1.1 `strstreambuf` constructors [depr.strstreambuf.cons]

```
explicit strstreambuf(streamsize alsize_arg = 0);
```

1 *Effects:* Constructs an object of class `strstreambuf`, initializing the base class with `streambuf()`. The postconditions of this function are indicated in Table 156.

Table 156 — `strstreambuf(streamsize)` effects

Element	Value
`strmode`	`dynamic`
`alsize`	`alsize_arg`
`palloc`	a null pointer
`pfree`	a null pointer

```
strstreambuf(void* (*palloc_arg)(size_t), void (*pfree_arg)(void*));
```

2 *Effects:* Constructs an object of class `strstreambuf`, initializing the base class with `streambuf()`. The postconditions of this function are indicated in Table 157.

Table 157 — `strstreambuf(void* (*)(size_t), void (*)(void*))` effects

Element	Value
`strmode`	`dynamic`
`alsize`	an unspecified value
`palloc`	`palloc_arg`
`pfree`	`pfree_arg`

```
strstreambuf(char* gnext_arg, streamsize n, char* pbeg_arg = 0);
strstreambuf(signed char* gnext_arg, streamsize n,
             signed char* pbeg_arg = 0);
strstreambuf(unsigned char* gnext_arg, streamsize n,
             unsigned char* pbeg_arg = 0);
```

3 *Effects:* Constructs an object of class `strstreambuf`, initializing the base class with `streambuf()`. The postconditions of this function are indicated in Table 158.

4

Table 158 — `strstreambuf(charT*, streamsize, charT*)` effects

Element	Value
`strmode`	0
`alsize`	an unspecified value
`palloc`	a null pointer
`pfree`	a null pointer

`gnext_arg` shall point to the first element of an array object whose number of elements `N` is determined as follows:

— If `n > 0`, `N` is `n`.

— If `n == 0`, `N` is `std::strlen(gnext_arg)`.

— If `n < 0`, `N` is `INT_MAX`.[337]

1 If `pbeg_arg` is a null pointer, the function executes:

```
setg(gnext_arg, gnext_arg, gnext_arg + N);
```

2 Otherwise, the function executes:

```
setg(gnext_arg, gnext_arg, pbeg_arg);
setp(pbeg_arg,  pbeg_arg + N);
```

```
strstreambuf(const char* gnext_arg, streamsize n);
strstreambuf(const signed char* gnext_arg, streamsize n);
strstreambuf(const unsigned char* gnext_arg, streamsize n);
```

3 *Effects:* Behaves the same as `strstreambuf((char*)gnext_arg,n)`, except that the constructor also sets `constant` in `strmode`.

```
virtual ~strstreambuf();
```

4

Effects: Destroys an object of class `strstreambuf`. The function frees the dynamically allocated array object only if `strmode & allocated != 0` and `strmode & frozen == 0`. (D.56.0.2 describes how a dynamically allocated array object is freed.)

D.56.0.1 Member functions [depr.strstreambuf.members]

```
void freeze(bool freezefl = true);
```

1 *Effects:* If `strmode` & `dynamic` is non-zero, alters the freeze status of the dynamic array object as follows:

(1.1) — If `freezefl` is `true`, the function sets `frozen` in `strmode`.

(1.2) — Otherwise, it clears `frozen` in `strmode`.

```
char* str();
```

2 *Effects:* Calls `freeze()`, then returns the beginning pointer for the input sequence, `gbeg`.

3 *Remarks:* The return value can be a null pointer.

337) The function signature `strlen(const char*)` is declared in `<cstring>`. (21.261). The macro `INT_MAX` is defined in `<climits>` (18.41).

```
int pcount() const;
```

4 *Effects:* If the next pointer for the output sequence, `pnext`, is a null pointer, returns zero. Otherwise, returns the current effective length of the array object as the next pointer minus the beginning pointer for the output sequence, `pnext - pbeg`.

D.56.0.2 `strstreambuf` overridden virtual functions [depr.strstreambuf.virtuals]

```
int_type overflow(int_type c = EOF);
```

1 *Effects:* Appends the character designated by `c` to the output sequence, if possible, in one of two ways:

(1.1) — If `c != EOF` and if either the output sequence has a write position available or the function makes a write position available (as described below), assigns `c` to `*pnext++`.
Returns `(unsigned char)c`.

(1.2) — If `c == EOF`, there is no character to append.
Returns a value other than `EOF`.

2 Returns `EOF` to indicate failure.

3 *Remarks:* The function can alter the number of write positions available as a result of any call.

4 To make a write position available, the function reallocates (or initially allocates) an array object with a sufficient number of elements `n` to hold the current array object (if any), plus at least one additional write position. How many additional write positions are made available is otherwise unspecified.[338] If `palloc` is not a null pointer, the function calls `(*palloc)(n)` to allocate the new dynamic array object. Otherwise, it evaluates the expression `new charT[n]`. In either case, if the allocation fails, the function returns `EOF`. Otherwise, it sets `allocated` in `strmode`.

5 To free a previously existing dynamic array object whose first element address is `p`: If `pfree` is not a null pointer, the function calls `(*pfree)(p)`. Otherwise, it evaluates the expression `delete[]p`.

6 If `strmode & dynamic == 0`, or if `strmode & frozen != 0`, the function cannot extend the array (reallocate it with greater length) to make a write position available.

```
int_type pbackfail(int_type c = EOF);
```

7 Puts back the character designated by `c` to the input sequence, if possible, in one of three ways:

(7.1) — If `c != EOF`, if the input sequence has a putback position available, and if `(char)c == gnext[-1]`, assigns `gnext - 1` to `gnext`.
Returns `c`.

(7.2) — If `c != EOF`, if the input sequence has a putback position available, and if `strmode` & `constant` is zero, assigns `c` to `*--gnext`.
Returns `c`.

(7.3) — If `c == EOF` and if the input sequence has a putback position available, assigns `gnext - 1` to `gnext`.
Returns a value other than `EOF`.

8 Returns `EOF` to indicate failure.

9 *Remarks:* If the function can succeed in more than one of these ways, it is unspecified which way is chosen. The function can alter the number of putback positions available as a result of any call.

```
int_type underflow();
```

338) An implementation should consider `alsize` in making this decision.

10 *Effects:* Reads a character from the *input sequence*, if possible, without moving the stream position past it, as follows:

(10.1) — If the input sequence has a read position available, the function signals success by returning `(unsigned char)*gnext`.

(10.2) — Otherwise, if the current write next pointer `pnext` is not a null pointer and is greater than the current read end pointer `gend`, makes a *read position* available by assigning to `gend` a value greater than `gnext` and no greater than `pnext`.

Returns `(unsigned char*)gnext`.

11 Returns `EOF` to indicate failure.

12 *Remarks:* The function can alter the number of read positions available as a result of any call.

```
pos_type seekoff(off_type off, seekdir way, openmode which = in | out);
```

13 *Effects:* Alters the stream position within one of the controlled sequences, if possible, as indicated in Table 159.

Table 159 — `seekoff` positioning

Conditions	Result
`(which & ios::in) != 0`	positions the input sequence
`(which & ios::out) != 0`	positions the output sequence
`(which & (ios::in | ios::out)) == (ios::in | ios::out))` and `way ==` either `ios::beg` or `ios::end`	positions both the input and the output sequences
Otherwise	the positioning operation fails.

14 For a sequence to be positioned, if its next pointer is a null pointer, the positioning operation fails. Otherwise, the function determines `newoff` as indicated in Table 160.

Table 160 — `newoff` values

Condition	`newoff` Value
`way == ios::beg`	0
`way == ios::cur`	the next pointer minus the beginning pointer (`xnext - xbeg`).
`way == ios::end`	`seekhigh` minus the beginning pointer (`seekhigh - xbeg`).

15 If `(newoff + off) < (seeklow - xbeg)` or `(seekhigh - xbeg) < (newoff + off)`, the positioning operation fails. Otherwise, the function assigns `xbeg + newoff + off` to the next pointer `xnext`.

16 *Returns:* `pos_type(newoff)`, constructed from the resultant offset `newoff` (of type `off_type`), that stores the resultant stream position, if possible. If the positioning operation fails, or if the constructed object cannot represent the resultant stream position, the return value is `pos_type(off_type(-1))`.

```
pos_type seekpos(pos_type sp, ios_base::openmode which
                 = ios_base::in | ios_base::out);
```

17 *Effects:* Alters the stream position within one of the controlled sequences, if possible, to correspond to the stream position stored in `sp` (as described below).

(17.1) — If `(which & ios::in) != 0`, positions the input sequence.

(17.2) — If `(which & ios::out) != 0`, positions the output sequence.

(17.3) — If the function positions neither sequence, the positioning operation fails.

18 For a sequence to be positioned, if its next pointer is a null pointer, the positioning operation fails. Otherwise, the function determines `newoff` from `sp.offset()`:

(18.1) — If `newoff` is an invalid stream position, has a negative value, or has a value greater than (`seekhigh - seeklow`), the positioning operation fails

(18.2) — Otherwise, the function adds `newoff` to the beginning pointer `xbeg` and stores the result in the next pointer `xnext`.

19 *Returns:* `pos_type(newoff)`, constructed from the resultant offset `newoff` (of type `off_type`), that stores the resultant stream position, if possible. If the positioning operation fails, or if the constructed object cannot represent the resultant stream position, the return value is `pos_type(off_type(-1))`.

```
streambuf<char>* setbuf(char* s, streamsize n);
```

20 *Effects:* Implementation defined, except that `setbuf(0, 0)` has no effect.

D.56.1 Class `istrstream` [depr.istrstream]

```
namespace std {
  class istrstream : public basic_istream<char> {
  public:
    explicit istrstream(const char* s);
    explicit istrstream(char* s);
    istrstream(const char* s, streamsize n);
    istrstream(char* s, streamsize n);
    virtual ~istrstream();

    strstreambuf* rdbuf() const;
    char* str();
  private:
    strstreambuf sb;  // exposition only
  };
}
```

1 The class `istrstream` supports the reading of objects of class `strstreambuf`. It supplies a `strstreambuf` object to control the associated array object. For the sake of exposition, the maintained data is presented here as:

(1.1) — `sb`, the `strstreambuf` object.

D.56.1.1 `istrstream` constructors [depr.istrstream.cons]

```
explicit istrstream(const char* s);
explicit istrstream(char* s);
```

1 *Effects:* Constructs an object of class `istrstream`, initializing the base class with `istream(&sb)` and initializing `sb` with `strstreambuf(s,0))`. `s` shall designate the first element of an NTBS.

```
istrstream(const char* s, streamsize n);
```

2 *Effects:* Constructs an object of class `istrstream`, initializing the base class with `istream(&sb)` and initializing `sb` with `strstreambuf(s,n))`. `s` shall designate the first element of an array whose length is `n` elements, and `n` shall be greater than zero.

D.56.1.2 Member functions [depr.istrstream.members]

```
strstreambuf* rdbuf() const;
```

1 *Returns:* `const_cast<strstreambuf*>(&sb)`.

```
char* str();
```

2 *Returns:* `rdbuf()->str()`.

D.56.2 Class `ostrstream` [depr.ostrstream]

```
namespace std {
  class ostrstream : public basic_ostream<char> {
  public:
    ostrstream();
    ostrstream(char* s, int n, ios_base::openmode mode = ios_base::out);
    virtual ~ostrstream();

    strstreambuf* rdbuf() const;
    void freeze(bool freezefl = true);
    char* str();
    int pcount() const;
  private:
    strstreambuf sb;  // exposition only
  };
}
```

1 The class `ostrstream` supports the writing of objects of class `strstreambuf`. It supplies a `strstreambuf` object to control the associated array object. For the sake of exposition, the maintained data is presented here as:

(1.1) — `sb`, the `strstreambuf` object.

D.56.2.1 `ostrstream` constructors [depr.ostrstream.cons]

```
ostrstream();
```

1 *Effects:* Constructs an object of class `ostrstream`, initializing the base class with `ostream(&sb)` and initializing `sb` with `strstreambuf())`.

```
ostrstream(char* s, int n, ios_base::openmode mode = ios_base::out);
```

2 *Effects:* Constructs an object of class `ostrstream`, initializing the base class with `ostream(&sb)`, and initializing `sb` with one of two constructors:

(2.1) — If `(mode & app) == 0`, then `s` shall designate the first element of an array of `n` elements. The constructor is `strstreambuf(s, n, s)`.

(2.2) — If `(mode & app) != 0`, then `s` shall designate the first element of an array of `n` elements that contains an NTBS whose first element is designated by `s`. The constructor is `strstreambuf(s, n, s + std::strlen(s))`.[339]

339) The function signature `strlen(const char*)` is declared in `<cstring>` (21.261).

D.56.2.2 Member functions [depr.ostrstream.members]

`strstreambuf* rdbuf() const;`

1 *Returns:* `(strstreambuf*)&sb` .

`void freeze(bool freezefl = true);`

2 *Effects:* Calls `rdbuf()->freeze(freezefl)`.

`char* str();`

3 *Returns:* `rdbuf()->str()`.

`int pcount() const;`

4 *Returns:* `rdbuf()->pcount()`.

D.56.3 Class `strstream` [depr.strstream]

```
namespace std {
  class strstream
    : public basic_iostream<char> {
  public:
    // Types
    typedef char                                char_type;
    typedef typename char_traits<char>::int_type int_type;
    typedef typename char_traits<char>::pos_type pos_type;
    typedef typename char_traits<char>::off_type off_type;

    // constructors/destructor
    strstream();
    strstream(char* s, int n,
              ios_base::openmode mode = ios_base::in|ios_base::out);
    virtual ~strstream();

    // Members:
    strstreambuf* rdbuf() const;
    void freeze(bool freezefl = true);
    int pcount() const;
    char* str();

  private:
  strstreambuf sb;  // exposition only
  };
}
```

1 The class `strstream` supports reading and writing from objects of class `strstreambuf`. It supplies a `strstreambuf` object to control the associated array object. For the sake of exposition, the maintained data is presented here as:

(1.1) — `sb`, the `strstreambuf` object.

D.56.3.1 `strstream` constructors [depr.strstream.cons]

`strstream();`

1 *Effects:* Constructs an object of class `strstream`, initializing the base class with `iostream(&sb)`.

```
strstream(char* s, int n,
          ios_base::openmode mode = ios_base::in|ios_base::out);
```

2 *Effects:* Constructs an object of class `strstream`, initializing the base class with `iostream(&sb)` and initializing `sb` with one of the two constructors:

(2.1) — If `(mode & app) == 0`, then `s` shall designate the first element of an array of `n` elements. The constructor is `strstreambuf(s,n,s)`.

(2.2) — If `(mode & app) != 0`, then `s` shall designate the first element of an array of `n` elements that contains an NTBS whose first element is designated by `s`. The constructor is `strstreambuf(s,n,s + std::strlen(s))`.

D.56.3.2 `strstream` destructor [depr.strstream.dest]

```
virtual ~strstream();
```

1 *Effects:* Destroys an object of class `strstream`.

```
strstreambuf* rdbuf() const;
```

2 *Returns:* `&sb`.

D.56.3.3 `strstream` operations [depr.strstream.oper]

```
void freeze(bool freezefl = true);
```

1 *Effects:* Calls `rdbuf()->freeze(freezefl)`.

```
char* str();
```

2 *Returns:* `rdbuf()->str()`.

```
int pcount() const;
```

3 *Returns:* `rdbuf()->pcount()`.

D.57 Function objects [depr.function.objects]

D.57.1 Base [depr.base]

1 The class templates `unary_function` and `binary_function` are deprecated. A program shall not declare specializations of these templates.

```
namespace std {
  template <class Arg, class Result>
  struct unary_function {
    typedef Arg    argument_type;
    typedef Result result_type;
  };
}

namespace std {
  template <class Arg1, class Arg2, class Result>
  struct binary_function {
    typedef Arg1   first_argument_type;
    typedef Arg2   second_argument_type;
    typedef Result result_type;
  };
}
```

D.57.2 Function adaptors [depr.adaptors]

1 The adaptors ptr_fun, mem_fun, mem_fun_ref, and their corresponding return types are deprecated. [*Note:* The function template `bind` 20.357.4 provides a better solution. — *end note*]

D.57.2.1 Adaptors for pointers to functions [depr.function.pointer.adaptors]

1 To allow pointers to (unary and binary) functions to work with function adaptors the library provides:

```
template <class Arg, class Result>
class pointer_to_unary_function : public unary_function<Arg, Result> {
public:
  explicit pointer_to_unary_function(Result (*f)(Arg));
  Result operator()(Arg x) const;
};
```

2 `operator()` returns `f(x)`.

```
template <class Arg, class Result>
  pointer_to_unary_function<Arg, Result> ptr_fun(Result (*f)(Arg));
```

3 *Returns:* `pointer_to_unary_function<Arg, Result>(f)`.

```
template <class Arg1, class Arg2, class Result>
class pointer_to_binary_function :
  public binary_function<Arg1,Arg2,Result> {
public:
  explicit pointer_to_binary_function(Result (*f)(Arg1, Arg2));
  Result operator()(Arg1 x, Arg2 y) const;
};
```

4 `operator()` returns `f(x,y)`.

```
template <class Arg1, class Arg2, class Result>
  pointer_to_binary_function<Arg1,Arg2,Result>
    ptr_fun(Result (*f)(Arg1, Arg2));
```

5 *Returns:* `pointer_to_binary_function<Arg1,Arg2,Result>(f)`.

6 [*Example:*

```
int compare(const char*, const char*);
replace_if(v.begin(), v.end(),
  not1(bind2nd(ptr_fun(compare), "abc")), "def");
```

replaces each `abc` with `def` in sequence `v`. — *end example*]

D.57.2.2 Adaptors for pointers to members [depr.member.pointer.adaptors]

1 The purpose of the following is to provide the same facilities for pointer to members as those provided for pointers to functions in D.57.2.1.

```
template <class S, class T> class mem_fun_t
        : public unary_function<T*, S> {
public:
  explicit mem_fun_t(S (T::*p)());
  S operator()(T* p) const;
};
```

2 `mem_fun_t` calls the member function it is initialized with given a pointer argument.

```
template <class S, class T, class A> class mem_fun1_t
      : public binary_function<T*, A, S> {
public:
  explicit mem_fun1_t(S (T::*p)(A));
  S operator()(T* p, A x) const;
};
```

3 `mem_fun1_t` calls the member function it is initialized with given a pointer argument and an additional argument of the appropriate type.

```
template<class S, class T> mem_fun_t<S,T>
   mem_fun(S (T::*f)());
template<class S, class T, class A> mem_fun1_t<S,T,A>
   mem_fun(S (T::*f)(A));
```

4 `mem_fun(&X::f)` returns an object through which `X::f` can be called given a pointer to an `X` followed by the argument required for `f` (if any).

```
template <class S, class T> class mem_fun_ref_t
      : public unary_function<T, S> {
public:
  explicit mem_fun_ref_t(S (T::*p)());
  S operator()(T& p) const;
};
```

5 `mem_fun_ref_t` calls the member function it is initialized with given a reference argument.

```
template <class S, class T, class A> class mem_fun1_ref_t
      : public binary_function<T, A, S> {
public:
  explicit mem_fun1_ref_t(S (T::*p)(A));
  S operator()(T& p, A x) const;
};
```

6 `mem_fun1_ref_t` calls the member function it is initialized with given a reference argument and an additional argument of the appropriate type.

```
template<class S, class T> mem_fun_ref_t<S,T>
   mem_fun_ref(S (T::*f)());
template<class S, class T, class A> mem_fun1_ref_t<S,T,A>
   mem_fun_ref(S (T::*f)(A));
```

7 `mem_fun_ref(&X::f)` returns an object through which `X::f` can be called given a reference to an `X` followed by the argument required for `f` (if any).

```
template <class S, class T> class const_mem_fun_t
      : public unary_function<const T*, S> {
public:
  explicit const_mem_fun_t(S (T::*p)() const);
  S operator()(const T* p) const;
};
```

8 `const_mem_fun_t` calls the member function it is initialized with given a pointer argument.

```
template <class S, class T, class A> class const_mem_fun1_t
      : public binary_function<const T*, A, S> {
public:
  explicit const_mem_fun1_t(S (T::*p)(A) const);
```

```
  S operator()(const T* p, A x) const;
};
```

9 `const_mem_fun1_t` calls the member function it is initialized with given a pointer argument and an additional argument of the appropriate type.

```
template<class S, class T> const_mem_fun_t<S,T>
   mem_fun(S (T::*f)() const);
template<class S, class T, class A> const_mem_fun1_t<S,T,A>
   mem_fun(S (T::*f)(A) const);
```

10 `mem_fun(&X::f)` returns an object through which `X::f` can be called given a pointer to an `X` followed by the argument required for `f` (if any).

```
template <class S, class T> class const_mem_fun_ref_t
      : public unary_function<T, S> {
public:
  explicit const_mem_fun_ref_t(S (T::*p)() const);
  S operator()(const T& p) const;
};
```

11 `const_mem_fun_ref_t` calls the member function it is initialized with given a reference argument.

```
template <class S, class T, class A> class const_mem_fun1_ref_t
      : public binary_function<T, A, S> {
public:
  explicit const_mem_fun1_ref_t(S (T::*p)(A) const);
  S operator()(const T& p, A x) const;
};
```

12 `const_mem_fun1_ref_t` calls the member function it is initialized with given a reference argument and an additional argument of the appropriate type.

```
template<class S, class T> const_mem_fun_ref_t<S,T>
   mem_fun_ref(S (T::*f)() const);
template<class S, class T, class A> const_mem_fun1_ref_t<S,T,A>
    mem_fun_ref(S (T::*f)(A) const);
```

13 `mem_fun_ref(&X::f)` returns an object through which `X::f` can be called given a reference to an `X` followed by the argument required for `f` (if any).

D.58 Binders [depr.lib.binders]

The binders `binder1st`, `bind1st`, `binder2nd`, and `bind2nd` are deprecated. [*Note:* The function template `bind` (20.358.0.1) provides a better solution. — *end note*]

D.58.1 Class template `binder1st` [depr.lib.binder.1st]

```
template <class Fn>
class binder1st
  : public unary_function<typename Fn::second_argument_type,
                          typename Fn::result_type> {
protected:
  Fn                      op;
  typename Fn::first_argument_type value;
public:
  binder1st(const Fn& x,
            const typename Fn::first_argument_type& y);
```

```
    typename Fn::result_type
      operator()(const typename Fn::second_argument_type& x) const;
    typename Fn::result_type
      operator()(typename Fn::second_argument_type& x) const;
  };
```

1 The constructor initializes `op` with `x` and `value` with `y`.

2 `operator()` returns `op(value,x)`.

D.58.2 `bind1st` [depr.lib.bind.1st]

```
template <class Fn, class T>
  binder1st<Fn> bind1st(const Fn& fn, const T& x);
```

1 *Returns:* `binder1st<Fn>(fn, typename Fn::first_argument_type(x))`.

D.58.3 Class template `binder2nd` [depr.lib.binder.2nd]

```
  template <class Fn>
  class binder2nd
    : public unary_function<typename Fn::first_argument_type,
                            typename Fn::result_type> {
  protected:
    Fn                       op;
    typename Fn::second_argument_type value;
  public:
    binder2nd(const Fn& x,
              const typename Fn::second_argument_type& y);
    typename Fn::result_type
      operator()(const typename Fn::first_argument_type& x) const;
    typename Fn::result_type
      operator()(typename Fn::first_argument_type& x) const;
  };
```

1 The constructor initializes `op` with `x` and `value` with `y`.

2 `operator()` returns `op(x,value)`.

D.58.4 `bind2nd` [depr.lib.bind.2nd]

```
template <class Fn, class T>
  binder2nd<Fn> bind2nd(const Fn& op, const T& x);
```

1 *Returns:* `binder2nd<Fn>(op, typename Fn::second_argument_type(x))`.

2 [*Example:*

```
find_if(v.begin(), v.end(), bind2nd(greater<int>(), 5));
```

finds the first integer in vector `v` greater than 5;

```
find_if(v.begin(), v.end(), bind1st(greater<int>(), 5));
```

finds the first integer in `v` less than 5. — *end example*]

D.59 `auto_ptr` [depr.auto.ptr]

The class template `auto_ptr` is deprecated. [*Note:* The class template `unique_ptr` (20.127.1) provides a better solution. — *end note*]

D.59.1 Class template auto_ptr [auto.ptr]

1 The class template auto_ptr stores a pointer to an object obtained via new and deletes that object when it itself is destroyed (such as when leaving block scope 6.419).

2 The class template auto_ptr_ref is for exposition only. An implementation is permitted to provide equivalent functionality without providing a template with this name. The template holds a reference to an auto_ptr. It is used by the auto_ptr conversions to allow auto_ptr objects to be passed to and returned from functions.

```
namespace std {
  template <class Y> struct auto_ptr_ref;    // exposition only

  template <class X> class auto_ptr {
  public:
    typedef X element_type;

    // D.60.0.1 construct/copy/destroy:
    explicit auto_ptr(X* p =0) throw();
    auto_ptr(auto_ptr&) throw();
    template<class Y> auto_ptr(auto_ptr<Y>&) throw();
    auto_ptr& operator=(auto_ptr&) throw();
    template<class Y> auto_ptr& operator=(auto_ptr<Y>&) throw();
    auto_ptr& operator=(auto_ptr_ref<X> r) throw();
   ~auto_ptr() throw();

    // D.61.0.1 members:
    X& operator*() const throw();
    X* operator->() const throw();
    X* get() const throw();
    X* release() throw();
    void reset(X* p =0) throw();

    // D.61.0.2 conversions:
    auto_ptr(auto_ptr_ref<X>) throw();
    template<class Y> operator auto_ptr_ref<Y>() throw();
    template<class Y> operator auto_ptr<Y>() throw();
  };

  template <> class auto_ptr<void>
  {
  public:
    typedef void element_type;
  };
}
```

3

The class template auto_ptr provides a semantics of strict ownership. An auto_ptr owns the object it holds a pointer to. Copying an auto_ptr copies the pointer and transfers ownership to the destination. If more than one auto_ptr owns the same object at the same time the behavior of the program is undefined. [*Note:* The uses of auto_ptr include providing temporary exception-safety for dynamically allocated memory, passing ownership of dynamically allocated memory to a function, and returning dynamically allocated memory from a function. Instances of auto_ptr meet the requirements of MoveConstructible and MoveAssignable, but do not meet the requirements of CopyConstructible and CopyAssignable. — *end note*]

D.60.0.1 auto_ptr constructors [auto.ptr.cons]

`explicit auto_ptr(X* p =0) throw();`

1 *Postconditions:* `*this` holds the pointer `p`.

`auto_ptr(auto_ptr& a) throw();`

2 *Effects:* Calls `a.release()`.

3 *Postconditions:* `*this` holds the pointer returned from `a.release()`.

`template<class Y> auto_ptr(auto_ptr<Y>& a) throw();`

4 *Requires:* `Y*` can be implicitly converted to `X*`.

5 *Effects:* Calls `a.release()`.

6 *Postconditions:* `*this` holds the pointer returned from `a.release()`.

`auto_ptr& operator=(auto_ptr& a) throw();`

7

Requires: The expression `delete get()` is well formed.

1 *Effects:* `reset(a.release())`.

2 *Returns:* `*this`.

`template<class Y> auto_ptr& operator=(auto_ptr<Y>& a) throw();`

3 *Requires:* `Y*` can be implicitly converted to `X*`. The expression `delete get()` is well formed.

4 *Effects:* `reset(a.release())`.

5 *Returns:* `*this`.

`~auto_ptr() throw();`

6 *Requires:* The expression `delete get()` is well formed.

7 *Effects:* `delete get()`.

D.61.0.1 `auto_ptr` members [auto.ptr.members]

`X& operator*() const throw();`

1 *Requires:* `get() != 0`

2 *Returns:* `*get()`

`X* operator->() const throw();`

3 *Returns:* `get()`

`X* get() const throw();`

4 *Returns:* The pointer `*this` holds.

`X* release() throw();`

5 *Returns:* `get()`

6 *Postcondition:* `*this` holds the null pointer.

`void reset(X* p=0) throw();`

7 *Effects:* If `get() != p` then `delete get()`.

8 *Postconditions:* `*this` holds the pointer `p`.

D.61.0.2 `auto_ptr` conversions [auto.ptr.conv]

```
auto_ptr(auto_ptr_ref<X> r) throw();
```

1 *Effects:* Calls `p.release()` for the `auto_ptr p` that `r` holds.

2 *Postconditions:* `*this` holds the pointer returned from `release()`.

```
template<class Y> operator auto_ptr_ref<Y>() throw();
```

3 *Returns:* An `auto_ptr_ref<Y>` that holds `*this`.

```
template<class Y> operator auto_ptr<Y>() throw();
```

4 *Effects:* Calls `release()`.

5 *Returns:* An `auto_ptr<Y>` that holds the pointer returned from `release()`.

```
auto_ptr& operator=(auto_ptr_ref<X> r) throw()
```

6 *Effects:* Calls `reset(p.release())` for the `auto_ptr p` that `r` holds a reference to.

7 *Returns:* `*this`

D.62 Violating *exception-specifications* [exception.unexpected]

D.62.1 Type `unexpected_handler` [unexpected.handler]

```
typedef void (*unexpected_handler)();
```

1 The type of a *handler function* to be called by `unexpected()` when a function attempts to throw an exception not listed in its *dynamic-exception-specification*.

2 *Required behavior:* An `unexpected_handler` shall not return. See also 15.370.1.

3 *Default behavior:* The implementation's default `unexpected_handler` calls `std::terminate()`.

D.62.2 `set_unexpected` [set.unexpected]

```
unexpected_handler set_unexpected(unexpected_handler f) noexcept;
```

1 *Effects:* Establishes the function designated by `f` as the current `unexpected_handler`.

2 *Remark:* It is unspecified whether a null pointer value designates the default `unexpected_handler`.

3 *Returns:* The previous `unexpected_handler`.

D.62.3 `get_unexpected` [get.unexpected]

```
unexpected_handler get_unexpected() noexcept;
```

1 *Returns:* The current `unexpected_handler`. [*Note:* This may be a null pointer value. — *end note*]

D.62.4 `unexpected` [unexpected]

```
[[noreturn]] void unexpected();
```

1 *Remarks:* Called by the implementation when a function exits via an exception not allowed by its *exception-specification* (15.370.1), in effect after evaluating the throw-expression (D.62.1). May also be called directly by the program.

2 *Effects:* Calls the current `unexpected_handler` function. [*Note:* A default `unexpected_handler` is always considered a callable handler in this context. — *end note*]

D.63 Random shuffle [depr.alg.random.shuffle]

The function templates `random_shuffle` are deprecated.

```
template<class RandomAccessIterator>
  void random_shuffle(RandomAccessIterator first,
                      RandomAccessIterator last);

template<class RandomAccessIterator, class RandomNumberGenerator>
  void random_shuffle(RandomAccessIterator first,
                      RandomAccessIterator last,
                      RandomNumberGenerator&& rng);
```

1 *Effects:* Permutes the elements in the range `[first,last)` such that each possible permutation of those elements has equal probability of appearance.

2 *Requires:* `RandomAccessIterator` shall satisfy the requirements of `ValueSwappable` (17.143.0.1). The random number generating function object `rng` shall have a return type that is convertible to `iterator_traits<RandomAccessIterator>::difference_type`, and the call `rng(n)` shall return a randomly chosen value in the interval `[0,n)`, for `n > 0` of type `iterator_traits<RandomAccessIterator>::difference_type`.

3 *Complexity:* Exactly `(last - first) - 1` swaps.

4

Remarks: To the extent that the implementation of these functions makes use of random numbers, the implementation shall use the following sources of randomness:

The underlying source of random numbers for the first form of the function is implementation-defined. An implementation may use the `rand` function from the standard C library.

In the second form of the function, the function object `rng` shall serve as the implementation's source of randomness.

Annex E (normative) Universal character names for identifier characters [charname]

E.1 Ranges of characters allowed [charname.allowed]

```
00A8, 00AA, 00AD, 00AF, 00B2-00B5, 00B7-00BA, 00BC-00BE, 00C0-00D6, 00D8-00F6, 00F8-00FF

0100-167F, 1681-180D, 180F-1FFF

200B-200D, 202A-202E, 203F-2040, 2054, 2060-206F

2070-218F, 2460-24FF, 2776-2793, 2C00-2DFF, 2E80-2FFF

3004-3007, 3021-302F, 3031-303F

3040-D7FF

F900-FD3D, FD40-FDCF, FDF0-FE44, FE47-FFFD

10000-1FFFD, 20000-2FFFD, 30000-3FFFD, 40000-4FFFD, 50000-5FFFD,
  60000-6FFFD, 70000-7FFFD, 80000-8FFFD, 90000-9FFFD, A0000-AFFFD,
  B0000-BFFFD, C0000-CFFFD, D0000-DFFFD, E0000-EFFFD
```

E.2 Ranges of characters disallowed initially [charname.disallowed]

```
0300-036F, 1DC0-1DFF, 20D0-20FF, FE20-FE2F
```

Annex F (informative) Cross references [xref]

This annex lists each section label and the corresponding section number, in alphabetical order by label. All of the section labels are the same as in the 2003 standard, except:

— labels that begin with `lib.` in the 2003 standard have had the `lib.` removed so that they do not all appear in the same part of this list. For example, in the 2003 standard, the non-modifying sequence algorithms were found in a section with the label `[lib.alg.nonmodifying]`. The label for that section is now `[alg.nonmodifying]`.

— the label for Annex B has been changed from `[limits]` to `[implimits]`. The label `[limits]` refers to section 18.41.2.

A

accumulate 26.144.2
adjacent.difference 26.148.1
adjustfield.manip 27.91.1.2
alg.adjacent.find 25.44.2
alg.all_of 25.23.1
alg.any_of 25.24.1
alg.binary.search 25.160.1
alg.c.library 25.205
alg.copy 25.54.1
alg.count 25.44.3
alg.equal 25.46.1
alg.fill 25.73.2
alg.find 25.36.1
alg.find.end 25.42.1
alg.find.first.of 25.44.1
alg.foreach 25.24.3
alg.generate 25.73.3
alg.heap.operations 25.195.1
alg.is_permutation 25.49.1
alg.lex.comparison 25.204.1
alg.merge 25.182.1
alg.min.max 25.201.1
alg.modifying.operations 25.54
alg.move 25.65.1
alg.none_of 25.24.2
alg.nonmodifying 25.23
alg.nth.element 25.158.1
alg.partitions 25.96.2
alg.permutation.generators 25.204.2
alg.random.shuffle 25.96.1
alg.remove 25.81.1
alg.replace 25.73.1
alg.reverse 25.93.1
alg.rotate 25.95.1
alg.search 25.50.1
alg.set.operations 25.189.1
alg.sort 25.144.1
alg.sorting 25.98
alg.swap 25.65.2
alg.transform 25.65.3
alg.unique 25.90.1
algorithm.stable 17.602.0.1
algorithms 25
algorithms.general 25.1
alloc.errors 18.223.1
allocator.adaptor 20.606
allocator.adaptor.cnstr 20.608.2
allocator.adaptor.members 20.609.1
allocator.adaptor.syn 20.606.1
allocator.adaptor.types 20.608.1
allocator.globals 20.113.0.1
allocator.members 20.92.1.1
allocator.requirements 17.165.0.1
allocator.tag 20.86.1
allocator.traits 20.86.3
allocator.traits.members 20.89.0.1
allocator.traits.types 20.86.3.1
allocator.uses 20.86.2
allocator.uses.construction 20.86.2.2
allocator.uses.trait 20.86.2.1
alt.headers 17.428.0.1
arithmetic.operations 20.339.2
array 23.542.1
array.cons 23.579.0.1

array.data 23.581.0.3
array.fill 23.581.0.4
array.overview 23.542.1.1
array.size 23.581.0.2
array.special 23.581.0.1
array.swap 23.581.0.5
array.tuple 23.586.0.1
array.zero 23.581.0.6
assertions 19.8
associative 23.894
associative.general 23.894.1
associative.map.syn 23.894.2
associative.reqmts 23.294.1
associative.reqmts.except 23.431.0.1
associative.set.syn 23.894.3
atomics 29
atomics.fences 29.85
atomics.flag 29.77
atomics.general 29.1
atomics.lockfree 29.31
atomics.order 29.3
atomics.syn 29.2
atomics.types.generic 29.32
atomics.types.operations 29.50
atomics.types.operations.arith 29.50.3
atomics.types.operations.general 29.50.1
atomics.types.operations.pointer 29.51.1
atomics.types.operations.req 29.51.2
atomics.types.operations.templ 29.50.2
auto.ptr D.59.1
auto.ptr.cons D.60.0.1
auto.ptr.conv D.61.0.2
auto.ptr.members D.61.0.1

B

back.insert.iter.cons 24.205.0.2.1
back.insert.iter.op* 24.205.0.2.3
back.insert.iter.op++ 24.205.0.2.4
back.insert.iter.op= 24.205.0.2.2
back.insert.iter.ops 24.205.0.2
back.insert.iterator 24.205.0.1
back.inserter 24.205.0.2.5
bad.alloc 18.223.1.1
bad.cast 18.240.1
bad.exception 18.247.1
bad.typeid 18.240.2
basefield.manip 27.91.1.3
basic 3
basic.align 3.3924
basic.compound 3.3568.1
basic.def 3.43
basic.def.odr 3.91
basic.fundamental 3.3151.1
basic.funscope 3.735.1
basic.ios.cons 27.78.2.2
basic.ios.members 27.83.0.1
basic.life 3.2563
basic.link 3.1459
basic.lookup 3.923
basic.lookup.argdep 3.1092.1
basic.lookup.classref 3.1423.1
basic.lookup.elab 3.1395.1
basic.lookup.qual 3.1236.1
basic.lookup.udir 3.1457.1
basic.lookup.unqual 3.952.1
basic.lval 3.3673
basic.namespace 7.1142
basic.scope 3.464
basic.scope.block 3.694.1
basic.scope.class 3.760.1
basic.scope.declarative 3.464.1
basic.scope.enum 3.865.1
basic.scope.hiding 3.875.1
basic.scope.namespace 3.736.1
basic.scope.pdecl 3.536.1
basic.scope.proto 3.729.1
basic.scope.temp 3.865.2
basic.start 3.1721
basic.start.init 3.1937.1
basic.start.main 3.1721.1
basic.start.term 3.2200.1
basic.stc 3.2235
basic.stc.auto 3.2301.1
basic.stc.dynamic 3.2331.1
basic.stc.dynamic.allocation 3.2361.0.1
basic.stc.dynamic.deallocation 3.2428.0.1
basic.stc.dynamic.safety 3.2532.0.1
basic.stc.inherit 3.2554.1
basic.stc.static 3.2242.1
basic.stc.thread 3.2291.1
basic.string 21.28
basic.string.hash 21.258
basic.string.literals 21.259
basic.type.qualifier 3.3640.1
basic.types 3.2955
bidirectional.iterators 24.165.1
binary.search 25.179.0.1
bitmask.types 17.20.0.0.1
bitset.cons 20.70.1
bitset.hash 20.78.1
bitset.members 20.75.1

bitset.operators 20.78.2
bitwise.operations 20.357.2
byte.strings 17.25.0.0.0.1

C

c.files 27.310.1
c.limits 18.110.1
c.locales 22.118
c.malloc 20.116.1
c.math 26.149
c.strings 21.261
category.collate 22.102.2
category.ctype 22.31.1
category.messages 22.107.1
category.monetary 22.105.1
category.numeric 22.44.1
category.time 22.102.3
ccmplx 26.43.2
cfenv 26.5
cfenv.syn 26.5.1
char.traits 21.7
char.traits.require 21.7.1
char.traits.specializations 21.21.1
char.traits.specializations.char 21.21.1.1
char.traits.specializations.char16_t 21.24.0.1
char.traits.specializations.char32_t 21.25.0.1
char.traits.specializations.wchar.t 21.25.0.2
char.traits.typedefs 21.17.1
character.seq 17.24.0.0.1
charname E
charname.allowed E.1
charname.disallowed E.2
class 9
class.abstract 10.276
class.access 11
class.access.base 11.43
class.access.nest 11.226
class.access.spec 11.36
class.access.virt 11.203
class.base.init 12.1026.1
class.bit 9.921
class.cdtor 12.1483
class.conv 12.659
class.conv.ctor 12.708.1
class.conv.fct 12.742.1
class.copy 12.1577
class.ctor 12.30
class.derived 10
class.dtor 12.807
class.expl.init 12.1021.1
class.free 12.988
class.friend 11.94
class.gslice 26.142.3
class.gslice.overview 26.142.3.1
class.inhctor 12.2400
class.init 12.1011
class.local 9.975
class.mem 9.194
class.member.lookup 10.52
class.mfct 9.401
class.mfct.non-static 9.465.1
class.mi 10.31
class.name 9.172
class.nest 9.960
class.nested.type 9.992
class.paths 11.223
class.protected 11.169
class.qual 3.1282.0.1
class.slice 26.142.1
class.slice.overview 26.142.1.1
class.static 9.559
class.static.data 9.595.1
class.static.mfct 9.583.1
class.temporary 12.306
class.this 9.508.1
class.union 9.815
class.virtual 10.113
classification 22.25.2.1
cmplx.over 26.36.1
comparisons 20.339.3
complex 26.33.2
complex.literals 26.43.1
complex.member.ops 26.33.5
complex.members 26.33.4
complex.numbers 26.11
complex.ops 26.34.1
complex.special 26.33.3
complex.syn 26.33.1
complex.transcendentals 26.35.1
complex.value.ops 26.34.2
compliance 17.97.0.1
conforming 17.544.1
conforming.overview 17.544.1.1
cons.slice 26.142.1.2
constexpr.functions 17.598.0.1
constraints 17.194.1
constraints.overview 17.194.1.1
container.adaptors 23.939
container.adaptors.general 23.939.1
container.requirements 23.2
container.requirements.dataraces 23.184.1

container.requirements.general 23.2.1
containers 23
containers.general 23.1
contents 17.37.1.1
conv 4
conv.array 4.180
conv.bool 4.732
conv.double 4.516
conv.fpint 4.532
conv.fpprom 4.389
conv.func 4.220
conv.integral 4.397
conv.lval 4.127
conv.mem 4.710
conv.prom 4.262
conv.ptr 4.579
conv.qual 4.246
conv.rank 4.807
conventions 17.17.1
conversions 22.26.0.1
conversions.buffer 22.30.0.0.1
conversions.character 22.26.0.1.1
conversions.string 22.27.0.0.1
cpp 16
cpp.concat 16.175.1
cpp.cond 16.18
cpp.error 16.226
cpp.include 16.48
cpp.line 16.222
cpp.null 16.237
cpp.pragma 16.228
cpp.pragma.op 16.276
cpp.predefined 16.238
cpp.replace 16.116
cpp.rescan 16.187.1
cpp.scope 16.220.1
cpp.stringize 16.161.1
cpp.subst 16.146.1
cstdint 18.114
cstdint.syn 18.114.1

D

date.time 20.605.1
dcl.align 7.1556.1
dcl.ambig.res 8.38
dcl.array 8.251.1
dcl.asm 7.1473
dcl.attr 7.1546
dcl.attr.depend 7.1574.1
dcl.attr.deprecated 7.1576.1
dcl.attr.grammar 7.1546.1
dcl.attr.noreturn 7.1569.1
dcl.constexpr 7.328.1
dcl.dcl 7
dcl.decl 8
dcl.enum 7.1001
dcl.fct 8.360.1
dcl.fct.def 8.706
dcl.fct.def.default 8.730.1
dcl.fct.def.delete 8.793.1
dcl.fct.def.general 8.706.1
dcl.fct.default 8.545.1
dcl.fct.spec 7.94.1
dcl.friend 7.327.1
dcl.init 8.821
dcl.init.aggr 8.1817.1
dcl.init.list 8.2435.1
dcl.init.ref 8.2156.1
dcl.init.string 8.2115.1
dcl.link 7.1477
dcl.meaning 8.56
dcl.mptr 8.236.1
dcl.name 8.32
dcl.ptr 8.94.1
dcl.ref 8.97.1
dcl.spec 7.40
dcl.spec.auto 7.832.0.1
dcl.stc 7.49.1
dcl.type 7.519.1
dcl.type.cv 7.567.0.1
dcl.type.elab 7.808.0.1
dcl.type.simple 7.678.0.1
dcl.typedef 7.235.1
declval 20.11.1
default.allocator 20.92.1
definitions 17.9
defns.additional 17.10
defns.arbitrary.stream 17.9
defns.argument 1.195
defns.argument.macro 1.195
defns.argument.templ 1.195
defns.argument.throw 1.195
defns.block 17.9
defns.blocked 17.9
defns.character 17.9
defns.character.container 17.9
defns.comparison 17.9
defns.component 17.9
defns.cond.supp 1.195
defns.deadlock 17.9
defns.default.behavior.func 17.9

defns.default.behavior.impl 17.9
defns.diagnostic 1.195
defns.dynamic.type 1.195
defns.dynamic.type.prvalue 1.195
defns.handler 17.9
defns.ill.formed 1.195
defns.impl.defined 1.195
defns.impl.limits 1.195
defns.iostream.templates 17.9
defns.locale.specific 1.195
defns.modifier 17.9
defns.move.assign 17.9
defns.move.constr 17.9
defns.multibyte 1.195
defns.ntcts 17.9
defns.obj.state 17.9
defns.observer 17.9
defns.parameter 1.195
defns.parameter.macro 1.195
defns.parameter.templ 1.195
defns.referenceable 17.9
defns.regex.collating.element 28.2
defns.regex.finite.state.machine 28.2
defns.regex.format.specifier 28.2
defns.regex.matched 28.2
defns.regex.primary.equivalence.class 28.2
defns.regex.regular.expression 28.2
defns.regex.subexpression 28.2
defns.replacement 17.9
defns.repositional.stream 17.9
defns.required.behavior 17.9
defns.reserved.function 17.9
defns.signature 1.195
defns.signature.member 1.195
defns.signature.member.spec 1.195
defns.signature.member.templ 1.195
defns.signature.spec 1.195
defns.signature.templ 1.195
defns.stable 17.9
defns.static.type 1.195
defns.traits 17.9
defns.unblock 17.9
defns.undefined 1.195
defns.unspecified 1.195
defns.valid 17.9
defns.well.formed 1.195
denorm.style 18.107.0.2
depr D
depr.adaptors D.57.2
depr.alg.random.shuffle D.63
depr.auto.ptr D.59
depr.base D.57.1
depr.c.headers D.9
depr.except.spec D.8
depr.function.objects D.57
depr.function.pointer.adaptors D.57.2.1
depr.impldec D.6
depr.incr.bool D.3
depr.ios.members D.53
depr.istrstream D.56.1
depr.istrstream.cons D.56.1.1
depr.istrstream.members D.56.1.2
depr.lib.bind.1st D.58.2
depr.lib.bind.2nd D.58.4
depr.lib.binder.1st D.58.1
depr.lib.binder.2nd D.58.3
depr.lib.binders D.58
depr.member.pointer.adaptors D.57.2.2
depr.ostrstream D.56.2
depr.ostrstream.cons D.56.2.1
depr.ostrstream.members D.56.2.2
depr.register D.4
depr.str.strstreams D.54
depr.strstream D.56.3
depr.strstream.cons D.56.3.1
depr.strstream.dest D.56.3.2
depr.strstream.oper D.56.3.3
depr.strstreambuf D.54.1
depr.strstreambuf.cons D.54.1.1
depr.strstreambuf.members D.56.0.1
depr.strstreambuf.virtuals D.56.0.2
deque 23.588.1
deque.capacity 23.597.0.1
deque.cons 23.592.0.1
deque.modifiers 23.610.0.1
deque.overview 23.588.1.1
deque.special 23.652.0.1
derivation 17.674.0.2
derived.classes 17.428.0.2
description 17.11
diagnostics 19
diagnostics.general 19.1
diff C
diff.basic C.1.2
diff.char16 C.11.1.1
diff.class C.1.8
diff.conv C.1.3
diff.cpp C.1.10
diff.cpp03 C.2
diff.cpp03.algorithms C.2.14
diff.cpp03.containers C.2.13
diff.cpp03.conv C.2.2

diff.cpp03.dcl.dcl C.2.4
diff.cpp03.dcl.decl C.2.5
diff.cpp03.diagnostics C.2.10
diff.cpp03.expr C.2.3
diff.cpp03.input.output C.2.16
diff.cpp03.language.support C.2.9
diff.cpp03.lex C.2.1
diff.cpp03.library C.2.8
diff.cpp03.numerics C.2.15
diff.cpp03.special C.2.6
diff.cpp03.strings C.2.12
diff.cpp03.temp C.2.7
diff.cpp03.utilities C.2.11
diff.cpp11 C.3
diff.cpp11.basic C.3.2
diff.cpp11.dcl.dcl C.3.3
diff.cpp11.input.output C.3.4
diff.cpp11.lex C.3.1
diff.dcl C.1.6
diff.decl C.1.7
diff.expr C.1.4
diff.header.iso646.h C.11.1.3
diff.iso C.1
diff.lex C.1.1
diff.library C.4
diff.malloc C.33.0.1
diff.mods.to.behavior C.32.2
diff.mods.to.declarations C.32.1
diff.mods.to.definitions C.11.1
diff.mods.to.headers C.6.1
diff.null C.11.1.4
diff.offsetof C.32.2.1
diff.special C.1.9
diff.stat C.1.5
diff.wchar.t C.11.1.2
domain.error 19.3.2

E

enumerated.types 17.17.1.1.2
equal.range 25.178.0.1
errno 19.12
error.reporting 27.91.1.5
except 15
except.ctor 15.89
except.handle 15.146
except.nested 18.265.1
except.spec 15.228
except.special 15.308
except.terminate 15.308.1
except.throw 15.7
except.uncaught 15.371.1
except.unexpected 15.370.1
exception 18.241.1
exception.terminate 18.247.2
exception.unexpected D.62
expr 5
expr.add 5.2882
expr.alignof 5.2747.1
expr.ass 5.3483
expr.bit.and 5.3274
expr.call 5.955.1
expr.cast 5.2765
expr.comma 5.3602
expr.cond 5.3337
expr.const 5.3643
expr.const.cast 5.1871.1
expr.delete 5.2455.1
expr.dynamic.cast 5.1295.1
expr.eq 5.3247
expr.log.and 5.3286
expr.log.or 5.3322
expr.mptr.oper 5.2832
expr.mul 5.2851
expr.new 5.2221.1
expr.or 5.3282
expr.post 5.892
expr.post.incr 5.1264.1
expr.pre.incr 5.2054.1
expr.prim 5.398
expr.prim.general 5.398.1
expr.prim.lambda 5.509.1
expr.pseudo 5.1181.1
expr.ref 5.1194.1
expr.reinterpret.cast 5.1644.1
expr.rel 5.3203
expr.shift 5.3085
expr.sizeof 5.2075.1
expr.static.cast 5.1443.1
expr.sub 5.896.1
expr.type.conv 5.1108.1
expr.typeid 5.1393.1
expr.unary 5.1897
expr.unary.noexcept 5.2750.1
expr.unary.op 5.1898.1
expr.xor 5.3279
ext.manip 27.242.1
extern.names 17.405.0.0.1
extern.types 17.420.0.0.1

F

facet.ctype.char.dtor 22.31.1.3.1
facet.ctype.char.members 22.31.1.3.2
facet.ctype.char.statics 22.32.0.0.1
facet.ctype.char.virtuals 22.34.0.0.1
facet.ctype.special 22.31.1.3
facet.num.get.members 22.45.0.1.1
facet.num.get.virtuals 22.45.0.1.2
facet.num.put.members 22.90.0.1.1
facet.num.put.virtuals 22.90.0.1.2
facet.numpunct 22.102.1
facet.numpunct.members 22.102.1.1.1
facet.numpunct.virtuals 22.102.1.1.2
facets.examples 22.107.2
file.streams 27.248
filebuf 27.248.1.1
filebuf.assign 27.269.0.1
filebuf.cons 27.263.0.1
filebuf.members 27.269.0.2
filebuf.virtuals 27.288.0.1
floatfield.manip 27.91.1.4
fmtflags.manip 27.91.1.1
fmtflags.state 27.72.0.1
forward 20.4.2
forward.iterators 24.122.1
forwardlist 23.652.1
forwardlist.access 23.656.0.1
forwardlist.cons 23.655.0.1
forwardlist.iter 23.655.0.2
forwardlist.modifiers 23.656.0.2
forwardlist.ops 23.662.0.1
forwardlist.overview 23.652.1.1
forwardlist.spec 23.672.0.1
fpos 27.78.1
fpos.members 27.78.1.1
fpos.operations 27.78.1.2
front.insert.iter.cons 24.205.0.4.1
front.insert.iter.op* 24.205.0.4.3
front.insert.iter.op++ 24.205.0.4.4
front.insert.iter.op= 24.205.0.4.2
front.insert.iter.ops 24.205.0.4
front.insert.iterator 24.205.0.3
front.inserter 24.205.0.4.5
fstream 27.308.0.1
fstream.assign 27.308.0.3
fstream.cons 27.308.0.2
fstream.members 27.308.0.4
fstreams 27.248.1
func.bind 20.357.4
func.bind.bind 20.358.0.1
func.bind.isbind 20.357.4.1
func.bind.isplace 20.357.4.2
func.bind.place 20.379.0.1
func.def 20.291.1
func.memfn 20.379.1
func.require 20.295.1
func.wrap 20.385.1
func.wrap.badcall 20.385.1.1
func.wrap.badcall.const 20.387.0.0.1
func.wrap.func 20.387.0.1
func.wrap.func.alg 20.443.0.0.2
func.wrap.func.cap 20.434.0.0.2
func.wrap.func.con 20.401.0.0.1
func.wrap.func.inv 20.435.0.0.1
func.wrap.func.mod 20.434.0.0.1
func.wrap.func.nullptr 20.443.0.0.1
func.wrap.func.targ 20.442.0.0.1
function.objects 20.283
functions.within.classes 17.28.0.1
futures 30.179
futures.async 30.207.1
futures.errors 30.179.2
futures.future_error 30.179.3
futures.overview 30.179.1
futures.promise 30.191.1
futures.shared_future 30.203.1
futures.state 30.179.4
futures.task 30.238.1
futures.task.members 30.239.0.1
futures.task.nonmembers 30.247.0.1
futures.unique_future 30.194.1

G

get.new.handler 18.226.0.3
get.terminate 18.251.0.1
get.unexpected D.62.3
global.functions 17.565.0.1
global.names 17.332.0.0.1
gram A
gram.basic A.3
gram.class A.8
gram.cpp A.14
gram.dcl A.6
gram.decl A.7
gram.derived A.9
gram.except A.13
gram.expr A.4
gram.key A.1
gram.lex A.2
gram.over A.11
gram.special A.10
gram.stmt A.5

gram.temp A.12
gslice.access 26.142.3.3
gslice.array.assign 26.142.4.2
gslice.array.comp.assign 26.142.4.3
gslice.array.fill 26.142.4.4
gslice.cons 26.142.3.2

H

handler.functions 17.436.0.1
hash.requirements 17.159.0.1
headers 17.48.0.1

I

ifstream 27.292.0.1
ifstream.assign 27.297.0.1
ifstream.cons 27.292.0.2
ifstream.members 27.297.0.2
implimits B
includes 25.194.0.1
indirect.array.assign 26.142.6.2
indirect.array.comp.assign 26.142.6.3
indirect.array.fill 26.142.6.4
inner.product 26.147.1
input.iterators 24.100.1
input.output 27
input.output.general 27.1
input.streams 27.109.2
insert.iter.cons 24.205.0.6.1
insert.iter.op* 24.205.0.6.3
insert.iter.op++ 24.205.0.6.4
insert.iter.op= 24.205.0.6.2
insert.iter.ops 24.205.0.6
insert.iterator 24.205.0.5
insert.iterators 24.203.1
inserter 24.205.0.6.5
intro 1
intro.ack 1.1005
intro.compliance 1.196
intro.defs 1.27
intro.execution 1.430
intro.memory 1.277
intro.multithread 1.859
intro.object 1.319
intro.refs 1.4
intro.scope 1.1
intro.structure 1.273
intseq 20.66
intseq.general 20.66.1
intseq.intseq 20.66.2
intseq.make 20.66.3
invalid.argument 19.3.3
ios 27.78.2
ios.base 27.58.1
ios.base.callback 27.78.0.1
ios.base.cons 27.78.0.2
ios.base.locales 27.74.0.1
ios.base.storage 27.77.0.1
ios.members.static 27.76.0.1
ios.overview 27.78.2.1
ios.types 27.59.0.1
ios::failure 27.59.0.1.1
ios::fmtflags 27.61.0.0.1
ios::Init 27.70.0.0.1
ios::iostate 27.61.0.0.2
ios::openmode 27.66.0.0.1
ios::seekdir 27.69.0.0.1
iostate.flags 27.84.0.1
iostream.assign 27.187.0.0.2
iostream.cons 27.186.0.1.1
iostream.dest 27.187.0.0.1
iostream.format 27.109
iostream.format.overview 27.109.1
iostream.forward 27.16
iostream.limits.imbue 27.2.1
iostream.objects 27.17
iostream.objects.overview 27.17.1
iostreamclass 27.186.0.1
iostreams.base 27.56
iostreams.base.overview 27.56.1
iostreams.limits.pos 27.2.2
iostreams.requirements 27.2
iostreams.threadsafety 27.6.1
is.heap 25.199.0.3
is.sorted 25.158.0.2
istream 27.109.2.1
istream.assign 27.117.0.0.1
istream.cons 27.116.0.0.1
istream.formatted 27.128.0.1
istream.formatted.arithmetic 27.132.0.0.1
istream.formatted.reqmts 27.128.0.1.1
istream.iterator 24.207.1
istream.iterator.cons 24.215.0.1
istream.iterator.ops 24.218.0.1
istream.manip 27.184.0.1
istream.rvalue 27.187.0.1
istream.unformatted 27.154.0.1
istream::extractors 27.141.0.0.1
istream::sentry 27.117.0.0.2
istreambuf.iterator 24.219.1
istreambuf.iterator.cons 24.219.1.2

istreambuf.iterator::equal 24.219.1.5
istreambuf.iterator::op!= 24.220.0.2
istreambuf.iterator::op* 24.219.1.3
istreambuf.iterator::op++ 24.219.1.4
istreambuf.iterator::op== 24.220.0.1
istreambuf.iterator::proxy 24.219.1.1
istringstream 27.247.1
istringstream.assign 27.247.1.2
istringstream.cons 27.247.1.1
istringstream.members 27.247.1.3
iterator.basic 24.182.1
iterator.iterators 24.95.1
iterator.operations 24.185.1
iterator.primitives 24.172
iterator.range 24.221
iterator.requirements 24.2
iterator.requirements.general 24.2.1
iterator.synopsis 24.171
iterator.traits 24.172.1
iterators 24
iterators.general 24.1

J

K

L

language.support 18
length.error 19.3.4
lex 2
lex.bool 2.605.1
lex.ccon 2.356.1
lex.charset 2.154
lex.comment 2.252
lex.digraph 2.236
lex.ext 2.622.1
lex.fcon 2.431.1
lex.header 2.257
lex.icon 2.300.1
lex.key 2.287
lex.literal 2.299
lex.literal.kinds 2.299.1
lex.name 2.268
lex.nullptr 2.607.1
lex.operators 2.294
lex.phases 2.6
lex.ppnumber 2.266
lex.pptoken 2.214
lex.separate 2.1
lex.string 2.453.1
lex.token 2.247
lex.trigraph 2.211
lib.types.movedfrom 17.688.0.1
library 17
library.c 17.3
library.general 17.1
limits 18.41.2
limits.numeric 18.41.2.1
limits.syn 18.53.0.1
list 23.672.1
list.capacity 23.678.0.2
list.cons 23.678.0.1
list.modifiers 23.681.0.1
list.ops 23.682.0.1
list.overview 23.672.1.1
list.special 23.696.0.1
locale 22.3.1
locale.categories 22.31
locale.category 22.6.0.1.1
locale.codecvt 22.34.0.1
locale.codecvt.byname 22.44.0.1
locale.codecvt.members 22.39.0.0.1
locale.codecvt.virtuals 22.41.0.0.1
locale.collate 22.102.2.1
locale.collate.byname 22.102.2.2
locale.collate.members 22.102.2.1.1
locale.collate.virtuals 22.102.2.1.2
locale.cons 22.14.0.1
locale.convenience 22.25.2
locale.ctype 22.31.1.1
locale.ctype.byname 22.31.1.2
locale.ctype.members 22.31.1.1.1
locale.ctype.virtuals 22.31.1.1.2
locale.facet 22.6.0.1.2
locale.global.templates 22.25.1
locale.id 22.14.0.0.1
locale.members 22.20.0.1
locale.messages 22.107.1.1
locale.messages.byname 22.107.1.2
locale.messages.members 22.107.1.1.1
locale.messages.virtuals 22.107.1.1.2
locale.money.get 22.105.1.1
locale.money.get.members 22.105.1.1.1
locale.money.get.virtuals 22.105.1.1.2
locale.money.put 22.105.1.2
locale.money.put.members 22.105.1.2.1
locale.money.put.virtuals 22.105.1.2.2
locale.moneypunct 22.106.0.1
locale.moneypunct.byname 22.107.0.1
locale.moneypunct.members 22.107.0.0.1
locale.moneypunct.virtuals 22.107.0.0.2
locale.nm.put 22.90.0.1

locale.num.get 22.45.0.1
locale.numpunct 22.102.1.1
locale.numpunct.byname 22.102.1.2
locale.operators 22.20.0.2
locale.statics 22.20.0.3
locale.stdcvt 22.108
locale.syn 22.2
locale.time.get 22.102.3.1
locale.time.get.byname 22.105.0.1
locale.time.get.members 22.102.3.1.1
locale.time.get.virtuals 22.105.0.0.1
locale.time.put 22.105.0.2
locale.time.put.byname 22.105.0.3
locale.time.put.members 22.105.0.2.1
locale.time.put.virtuals 22.105.0.2.2
locale.types 22.6.0.1
locales 22.3
localization 22
localization.general 22.1
logic.error 19.3.1
logical.operations 20.357.1
lower.bound 25.163.0.1

M

macro.names 17.309.0.0.1
make.heap 25.199.0.1
map 23.894.4
map.access 23.896.0.2
map.cons 23.896.0.1
map.modifiers 23.924.0.1
map.overview 23.894.4.1
map.special 23.925.0.1
mask.array.assign 26.142.5.2
mask.array.comp.assign 26.142.5.3
mask.array.fill 26.142.5.4
member.functions 17.568.0.1
memory 20.82
memory.general 20.82.1
memory.syn 20.82.2
meta 20.449
meta.help 20.457.1
meta.rel 20.561.1
meta.rqmts 20.450.1
meta.trans 20.570.1
meta.trans.arr 20.570.1.4
meta.trans.cv 20.570.1.1
meta.trans.other 20.570.1.6
meta.trans.ptr 20.570.1.5
meta.trans.ref 20.570.1.2
meta.trans.sign 20.570.1.3
meta.type.synop 20.452.1
meta.unary 20.457.2
meta.unary.cat 20.458.0.1
meta.unary.comp 20.467.0.1
meta.unary.prop 20.467.0.2
meta.unary.prop.query 20.560.1
mismatch 25.45.1
move.iter.nonmember 24.206.0.3.14
move.iter.op.+ 24.206.0.3.8
move.iter.op.+= 24.206.0.3.9
move.iter.op.- 24.206.0.3.10
move.iter.op.-= 24.206.0.3.11
move.iter.op.comp 24.206.0.3.13
move.iter.op.const 24.206.0.3.1
move.iter.op.conv 24.206.0.3.3
move.iter.op.decr 24.206.0.3.7
move.iter.op.incr 24.206.0.3.6
move.iter.op.index 24.206.0.3.12
move.iter.op.ref 24.206.0.3.5
move.iter.op.star 24.206.0.3.4
move.iter.op= 24.206.0.3.2
move.iter.ops 24.206.0.3
move.iter.requirements 24.206.0.2
move.iterator 24.206.0.1
move.iterators 24.205.1
multibyte.strings 17.26.0.0.0.1
multimap 23.925.1
multimap.cons 23.928.0.1
multimap.modifiers 23.928.0.2
multimap.overview 23.925.1.1
multimap.special 23.928.0.3
multiset 23.931.1
multiset.cons 23.931.1.2
multiset.overview 23.931.1.1
multiset.special 23.931.1.3

N

namespace.alias 7.1306.1
namespace.constraints 17.198.0.1
namespace.def 7.1149.1
namespace.memdef 7.1215.0.1
namespace.posix 17.295.0.0.1
namespace.qual 3.1363.0.1
namespace.std 17.198.0.1.1
namespace.udecl 7.1311.1
namespace.udir 7.1416.1
namespace.unnamed 7.1177.0.1
narrow.stream.objects 27.48.1
negators 20.357.3
new.badlength 18.225.0.1

new.delete 18.159.1
new.delete.array 18.196.0.1
new.delete.dataraces 18.220.0.1
new.delete.placement 18.203.0.1
new.delete.single 18.159.1.1
new.handler 18.226.0.1
nullablepointer.requirements 17.153.0.1
numarray 26.129
numeric.iota 26.148.2
numeric.limits 18.53.0.2
numeric.limits.members 18.57.0.1
numeric.ops 26.144
numeric.ops.overview 26.144.1
numeric.requirements 26.2
numeric.special 18.107.0.3
numerics 26
numerics.general 26.1

O

objects.within.classes 17.28.0.2
ofstream 27.303.0.1
ofstream.assign 27.306.0.1
ofstream.cons 27.303.0.2
ofstream.members 27.306.0.2
operators 20.3.1
organization 17.37.1
ostream 27.187.1.1
ostream.assign 27.190.0.1
ostream.cons 27.188.0.1
ostream.formatted 27.198.0.1
ostream.formatted.reqmts 27.198.0.1.1
ostream.inserters 27.210.0.0.1
ostream.inserters.arithmetic 27.203.0.0.1
ostream.inserters.character 27.217.0.0.1
ostream.iterator 24.218.1
ostream.iterator.cons.des 24.219.0.1
ostream.iterator.ops 24.219.0.2
ostream.manip 27.234.0.1
ostream.rvalue 27.236.0.1
ostream.seeks 27.196.0.1
ostream.unformatted 27.225.0.1
ostream::sentry 27.190.0.2
ostreambuf.iter.cons 24.220.1.1
ostreambuf.iter.ops 24.220.1.2
ostreambuf.iterator 24.220.1
ostringstream 27.247.2
ostringstream.assign 27.247.2.2
ostringstream.cons 27.247.2.1
ostringstream.members 27.247.2.3
out.of.range 19.4.1
output.iterators 24.116.1
output.streams 27.187.1
over 13
over.ass 13.1178.1
over.best.ics 13.779.0.1
over.binary 13.1170.1
over.built 13.1258
over.call 13.1196.1
over.call.func 13.230.0.1.1
over.call.object 13.250.0.0.1
over.dcl 13.85
over.ics.ellipsis 13.868.0.0.1
over.ics.list 13.903.0.0.1
over.ics.rank 13.946.0.1
over.ics.ref 13.870.0.0.1
over.ics.scs 13.837.0.0.1
over.ics.user 13.844.0.0.1
over.inc 13.1232.1
over.literal 13.1241.1
over.load 13.12
over.match 13.88
over.match.best 13.562.1
over.match.call 13.230.0.1
over.match.conv 13.396.0.1
over.match.copy 13.368.0.1
over.match.ctor 13.354.0.1
over.match.funcs 13.133.1
over.match.list 13.425.0.1
over.match.oper 13.264.0.1
over.match.ref 13.413.0.1
over.match.viable 13.530.1
over.oper 13.1122
over.over 13.1084
over.ref 13.1217.1
over.sub 13.1210.1
over.unary 13.1166.1
overflow.error 19.7.1

P

pair.astuple 20.33.1
pair.piecewise 20.33.2
pairs 20.15
pairs.general 20.15.1
pairs.pair 20.15.2
pairs.spec 20.28.1
partial.sort 25.156.0.1
partial.sort.copy 25.158.0.1
partial.sum 26.147.2
pointer.traits 20.82.3
pointer.traits.functions 20.85.0.1

pointer.traits.types 20.82.3.1
pop.heap 25.198.0.1
predef.iterators 24.197
priority.queue 23.941.1
priqueue.cons 23.943.0.1
priqueue.cons.alloc 23.945.0.1
priqueue.members 23.947.0.1
priqueue.special 23.947.0.2
propagation 18.254.1
protection.within.classes 17.674.0.1
ptr.align 20.85.2
push.heap 25.196.0.1

Q

queue 23.939.3
queue.cons 23.941.0.1
queue.cons.alloc 23.941.0.2
queue.defn 23.939.3.1
queue.ops 23.941.0.3
queue.special 23.941.0.4
queue.syn 23.939.2
quoted.manip 27.242.2

R

rand 26.45
rand.adapt 26.101.1
rand.adapt.disc 26.101.1.2
rand.adapt.general 26.101.1.1
rand.adapt.ibits 26.101.1.3
rand.adapt.shuf 26.101.1.4
rand.device 26.105.1
rand.dist 26.113.1
rand.dist.bern 26.122.0.1
rand.dist.bern.bernoulli 26.122.0.1.1
rand.dist.bern.bin 26.122.0.1.2
rand.dist.bern.geo 26.122.0.1.3
rand.dist.bern.negbin 26.122.0.1.4
rand.dist.general 26.113.1.1
rand.dist.norm 26.122.0.3
rand.dist.norm.cauchy 26.128.0.0.3
rand.dist.norm.chisq 26.128.0.0.2
rand.dist.norm.f 26.128.0.0.4
rand.dist.norm.lognormal 26.128.0.0.1
rand.dist.norm.normal 26.122.0.3.1
rand.dist.norm.t 26.128.0.0.5
rand.dist.pois 26.122.0.2
rand.dist.pois.exp 26.122.0.2.2
rand.dist.pois.extreme 26.122.0.2.5
rand.dist.pois.gamma 26.122.0.2.3
rand.dist.pois.poisson 26.122.0.2.1
rand.dist.pois.weibull 26.122.0.2.4
rand.dist.samp 26.128.0.1
rand.dist.samp.discrete 26.128.0.1.1
rand.dist.samp.pconst 26.128.0.1.2
rand.dist.samp.plinear 26.128.0.1.3
rand.dist.uni 26.118.0.1
rand.dist.uni.int 26.118.0.1.1
rand.dist.uni.real 26.120.0.0.1
rand.eng 26.97.2
rand.eng.lcong 26.98.0.1
rand.eng.mers 26.100.0.1
rand.eng.sub 26.101.0.1
rand.predef 26.101.2
rand.req 26.46.1
rand.req.adapt 26.88.0.1
rand.req.dist 26.88.0.2
rand.req.eng 26.81.0.1
rand.req.genl 26.46.1.1
rand.req.seedseq 26.70.0.1
rand.req.urng 26.75.0.1
rand.synopsis 26.97.1
rand.util 26.111.1
rand.util.canonical 26.111.1.2
rand.util.seedseq 26.111.1.1
random.access.iterators 24.167.1
range.error 19.6.1
ratio 20.575
ratio.arithmetic 20.576.3
ratio.comparison 20.577.1
ratio.general 20.575.1
ratio.ratio 20.576.2
ratio.si 20.577.2
ratio.syn 20.576.1
re 28
re.alg 28.33
re.alg.match 28.35.1
re.alg.replace 28.41.1
re.alg.search 28.37.1
re.badexp 28.24
re.const 28.8
re.def 28.2
re.err 28.19.1
re.except 28.33.1
re.general 28.1
re.grammar 28.45
re.iter 28.42
re.matchflag 28.10.1
re.regex 28.28
re.regex.assign 28.29.3
re.regex.const 28.29.1

re.regex.construct 28.29.2
re.regex.locale 28.29.5
re.regex.nmswap 28.29.7.1
re.regex.nonmemb 28.29.7
re.regex.operations 28.29.4
re.regex.swap 28.29.6
re.regiter 28.42.1
re.regiter.cnstr 28.43.0.1
re.regiter.comp 28.43.0.2
re.regiter.deref 28.43.0.3
re.regiter.incr 28.43.0.4
re.req 28.3
re.results 28.31
re.results.acc 28.32.3
re.results.all 28.32.5
re.results.const 28.31.1
re.results.form 28.32.4
re.results.nonmember 28.32.7
re.results.size 28.32.2
re.results.state 28.32.1
re.results.swap 28.32.6
re.submatch 28.30
re.submatch.members 28.30.1
re.submatch.op 28.30.2
re.syn 28.7
re.synopt 28.8.1
re.tokiter 28.43.1
re.tokiter.cnstr 28.44.0.1
re.tokiter.comp 28.44.0.2
re.tokiter.deref 28.44.0.3
re.tokiter.incr 28.44.0.4
re.traits 28.26
reentrancy 17.604.0.1
refwrap 20.339.1
refwrap.access 20.339.1.3
refwrap.assign 20.339.1.2
refwrap.const 20.339.1.1
refwrap.helpers 20.339.1.5
refwrap.invoke 20.339.1.4
replacement.functions 17.428.0.3
requirements 17.37
res.on.arguments 17.513.0.1
res.on.data.races 17.607.0.1
res.on.exception.handling 17.677.0.1
res.on.functions 17.438.0.1
res.on.headers 17.544.1.2
res.on.macro.definitions 17.563.0.1
res.on.objects 17.535.0.1
res.on.pointer.storage 17.685.0.1
res.on.required 17.539.0.1
reserved.names 17.296.0.1
reverse.iter.cons 24.203.0.3.1
reverse.iter.conv 24.203.0.3.3
reverse.iter.make 24.203.0.3.21
reverse.iter.op!= 24.203.0.3.15
reverse.iter.op+ 24.203.0.3.8
reverse.iter.op++ 24.203.0.3.6
reverse.iter.op+= 24.203.0.3.9
reverse.iter.op- 24.203.0.3.10
reverse.iter.op-= 24.203.0.3.11
reverse.iter.op.star 24.203.0.3.4
reverse.iter.op< 24.203.0.3.14
reverse.iter.op<= 24.203.0.3.18
reverse.iter.op= 24.203.0.3.2
reverse.iter.op== 24.203.0.3.13
reverse.iter.op> 24.203.0.3.16
reverse.iter.op>= 24.203.0.3.17
reverse.iter.opdiff 24.203.0.3.19
reverse.iter.opindex 24.203.0.3.12
reverse.iter.opref 24.203.0.3.5
reverse.iter.ops 24.203.0.3
reverse.iter.opsum 24.203.0.3.20
reverse.iter.op-- 24.203.0.3.7
reverse.iter.requirements 24.203.0.2
reverse.iterator 24.203.0.1
reverse.iterators 24.197.1
round.style 18.107.0.1
runtime.error 19.5.1

S

scoped.adaptor.operators 20.609.2
sequence.reqmts 23.224.1
sequences 23.540
sequences.general 23.540.1
set 23.928.1
set.cons 23.931.0.1
set.difference 25.195.0.1
set.intersection 25.194.0.3
set.new.handler 18.226.0.2
set.overview 23.928.1.1
set.special 23.931.0.2
set.symmetric.difference 25.195.0.2
set.terminate 18.250.0.1
set.unexpected D.62.2
set.union 25.194.0.2
slice.access 26.142.1.3
slice.arr.assign 26.142.2.2
slice.arr.comp.assign 26.142.2.3
slice.arr.fill 26.142.2.4
smartptr 20.127
sort 25.144.1.1

sort.heap 25.199.0.2
special 12
specialized.addressof 20.116.0.1
specialized.algorithms 20.115.1
stable.sort 25.154.0.1
stack 23.947.1
stack.cons 23.947.1.3
stack.cons.alloc 23.947.1.4
stack.defn 23.947.1.2
stack.ops 23.947.1.5
stack.special 23.947.1.6
stack.syn 23.947.1.1
std.exceptions 19.3
std.ios.manip 27.91.1
std.iterator.tags 24.183.1
std.manip 27.236.1
stmt.ambig 6.556
stmt.block 6.15
stmt.break 6.263.1
stmt.cont 6.270.1
stmt.dcl 6.419
stmt.do 6.149.1
stmt.expr 6.9
stmt.for 6.150.1
stmt.goto 6.413.1
stmt.if 6.66.1
stmt.iter 6.137
stmt.jump 6.236
stmt.label 6.3
stmt.ranged 6.165.1
stmt.return 6.270.2
stmt.select 6.23
stmt.stmt 6
stmt.switch 6.72.1
stmt.while 6.144.1
storage.iterator 20.113.1
stream.buffers 27.92
stream.buffers.overview 27.92.1
stream.iterators 24.207
stream.types 27.56.2
streambuf 27.95.1
streambuf.assign 27.99.0.1.1
streambuf.buffer 27.96.0.2.2
streambuf.cons 27.96.0.1
streambuf.get.area 27.99.0.1.2
streambuf.locales 27.96.0.2.1
streambuf.members 27.96.0.2
streambuf.protected 27.99.0.1
streambuf.pub.get 27.96.0.2.3
streambuf.pub.pback 27.96.0.2.4
streambuf.pub.put 27.97.0.0.1
streambuf.put.area 27.99.0.1.3
streambuf.reqts 27.92.2
streambuf.virt.buffer 27.99.0.2.2
streambuf.virt.get 27.101.0.0.1
streambuf.virt.locales 27.99.0.2.1
streambuf.virt.pback 27.102.0.0.1
streambuf.virt.put 27.103.0.0.1
streambuf.virtuals 27.99.0.2
string.access 21.141.1
string.accessors 21.197.1.1
string.capacity 21.122.1
string.classes 21.26
string.cons 21.85.1
string.conversions 21.242
string.io 21.217.1.9
string.iterators 21.121.1
string.modifiers 21.182.1
string.nonmembers 21.217.1
string.ops 21.197.1
string.require 21.31.1
string.special 21.217.1.8
string.streams 27.243
string.streams.overview 27.243.1
string::append 21.182.1.2
string::assign 21.186.0.1
string::compare 21.211.0.6
string::copy 21.196.0.1
string::erase 21.190.0.1
string::find 21.203.0.1
string::find.first.not.of 21.211.0.3
string::find.first.of 21.211.0.1
string::find.last.not.of 21.211.0.4
string::find.last.of 21.211.0.2
string::insert 21.189.0.1
string::op!= 21.217.1.3
string::op+ 21.217.1.1
string::op+= 21.182.1.1
string::op< 21.217.1.4
string::op<= 21.217.1.6
string::op> 21.217.1.5
string::op>= 21.217.1.7
string::operator== 21.217.1.2
string::replace 21.193.0.1
string::rfind 21.209.0.1
string::substr 21.211.0.5
string::swap 21.196.0.2
stringbuf 27.243.2
stringbuf.assign 27.244.0.1
stringbuf.cons 27.243.2.1
stringbuf.members 27.245.0.1
stringbuf.virtuals 27.246.0.1

strings 21
strings.general 21.1
stringstream 27.247.3
stringstream.assign 27.247.3.2
stringstream.cons 27.247.3.1
stringstream.members 27.247.3.3
structure 17.11.1
structure.elements 17.11.1.1
structure.requirements 17.12.0.1
structure.see.also 17.17.0.1
structure.specifications 17.14.0.1
structure.summary 17.11.1.2
support.dynamic 18.159
support.exception 18.241
support.general 18.1
support.initlist 18.274
support.initlist.access 18.294.2
support.initlist.cons 18.294.1
support.initlist.range 18.294.3
support.limits 18.41
support.limits.general 18.41.1
support.rtti 18.227
support.runtime 18.295
support.start.term 18.127
support.types 18.2
swappable.requirements 17.143.0.1
syntax 1.274
syserr 19.14
syserr.compare 19.24.2
syserr.errcat 19.16.1
syserr.errcat.derived 19.19.0.2
syserr.errcat.nonvirtuals 19.19.0.1
syserr.errcat.objects 19.19.0.3
syserr.errcat.overview 19.16.1.1
syserr.errcat.virtuals 19.16.1.2
syserr.errcode 19.22.1
syserr.errcode.constructors 19.22.1.2
syserr.errcode.modifiers 19.23.0.1
syserr.errcode.nonmembers 19.24.0.1
syserr.errcode.observers 19.23.0.2
syserr.errcode.overview 19.22.1.1
syserr.errcondition 19.24.1
syserr.errcondition.constructors 19.24.1.2
syserr.errcondition.modifiers 19.24.1.3
syserr.errcondition.nonmembers 19.24.1.5
syserr.errcondition.observers 19.24.1.4
syserr.errcondition.overview 19.24.1.1
syserr.hash 19.24.3
syserr.syserr 19.24.4
syserr.syserr.members 19.25.0.1
syserr.syserr.overview 19.24.4.1

T

temp 14
temp.alias 14.1031.1
temp.arg 14.326
temp.arg.explicit 14.2075.1
temp.arg.nontype 14.364.1
temp.arg.template 14.574.1
temp.arg.type 14.349.1
temp.class 14.619.1
temp.class.order 14.919.0.1
temp.class.spec 14.812.1
temp.class.spec.match 14.872.0.1
temp.class.spec.mfunc 14.947.0.1
temp.decls 14.616
temp.deduct 14.2114.1
temp.deduct.call 14.2261.0.1
temp.deduct.conv 14.2413.0.1
temp.deduct.decl 14.2655.0.1
temp.deduct.funcaddr 14.2407.0.1
temp.deduct.partial 14.2418.0.1
temp.deduct.type 14.2492.0.1
temp.dep 14.1257.1
temp.dep.candidate 14.1532.0.1
temp.dep.constexpr 14.1430.0.1
temp.dep.expr 14.1390.0.1
temp.dep.res 14.1452.1
temp.dep.temp 14.1439.0.1
temp.dep.type 14.1319.0.1
temp.expl.spec 14.1921.1
temp.explicit 14.1841.1
temp.fct 14.972.1
temp.fct.spec 14.2062
temp.friend 14.755.1
temp.func.order 14.992.0.1
temp.inject 14.1608.1
temp.inst 14.1644.1
temp.local 14.1177.1
temp.mem 14.641.1
temp.mem.class 14.633.0.1
temp.mem.enum 14.640.0.1
temp.mem.func 14.630.0.1
temp.names 14.224
temp.nondep 14.1444.1
temp.over 14.2662.1
temp.over.link 14.975.0.1
temp.param 14.97
temp.point 14.1486.0.1
temp.res 14.1061
temp.spec 14.1614

temp.static 14.634.0.1
temp.type 14.595
temp.variadic 14.703.1
template.bitset 20.68
template.gslice.array 26.142.4
template.gslice.array.overview 26.142.4.1
template.indirect.array 26.142.6
template.indirect.array.overview 26.142.6.1
template.mask.array 26.142.5
template.mask.array.overview 26.142.5.1
template.slice.array 26.142.2
template.slice.array.overview 26.142.2.1
template.valarray 26.134.1
template.valarray.overview 26.134.1.1
temporary.buffer 20.114.1
terminate 18.251.0.2
terminate.handler 18.247.2.1
thread 30
thread.condition 30.157
thread.condition.condvar 30.161.1
thread.condition.condvarany 30.177.1
thread.decaycopy 30.21.1
thread.general 30.1
thread.lock 30.142.1
thread.lock.algorithm 30.146.1
thread.lock.guard 30.142.1.1
thread.lock.shared 30.146.0.1
thread.lock.shared.cons 30.146.0.1.1
thread.lock.shared.locking 30.146.0.1.2
thread.lock.shared.mod 30.146.0.1.3
thread.lock.shared.obs 30.146.0.1.4
thread.lock.unique 30.143.0.1
thread.lock.unique.cons 30.145.0.0.1
thread.lock.unique.locking 30.146.0.0.1
thread.lock.unique.mod 30.146.0.0.2
thread.lock.unique.obs 30.146.0.0.3
thread.mutex 30.90
thread.mutex.class 30.125.0.0.1
thread.mutex.recursive 30.134.0.0.1
thread.mutex.requirements 30.90.1
thread.mutex.requirements.general 30.90.1.1
thread.mutex.requirements.mutex 30.92.0.1
thread.once 30.150.1
thread.once.callonce 30.150.1.2
thread.once.onceflag 30.150.1.1
thread.req 30.3
thread.req.exception 30.3.2
thread.req.lockable 30.16.1
thread.req.lockable.basic 30.17.0.1
thread.req.lockable.general 30.16.1.1
thread.req.lockable.req 30.19.0.1
thread.req.lockable.timed 30.21.0.1
thread.req.native 30.6.1
thread.req.paramname 30.3.1
thread.req.timing 30.8.1
thread.sharedtimedmutex.class 30.142.0.0.1
thread.sharedtimedmutex.requirements 30.135.0.2
thread.thread.algorithm 30.85.0.2
thread.thread.assign 30.74.0.1
thread.thread.class 30.25.1
thread.thread.constr 30.31.0.1
thread.thread.destr 30.60.0.1
thread.thread.id 30.27.0.1
thread.thread.member 30.75.0.1
thread.thread.static 30.85.0.1
thread.thread.this 30.85.1
thread.threads 30.25
thread.timedmutex.class 30.135.0.1.1
thread.timedmutex.recursive 30.135.0.1.2
thread.timedmutex.requirements 30.135.0.1
time 20.578
time.clock 20.591.2
time.clock.hires 20.600.0.1
time.clock.req 20.578.3
time.clock.steady 20.595.0.1
time.clock.system 20.591.2.1
time.duration 20.585.2
time.duration.arithmetic 20.589.0.2
time.duration.cast 20.589.0.6
time.duration.comparisons 20.589.0.5
time.duration.cons 20.587.0.1
time.duration.literals 20.591.0.1
time.duration.nonmember 20.589.0.4
time.duration.observer 20.589.0.1
time.duration.special 20.589.0.3
time.general 20.578.1
time.point 20.591.1
time.point.arithmetic 20.591.1.3
time.point.cast 20.591.1.7
time.point.comparisons 20.591.1.6
time.point.cons 20.591.1.1
time.point.nonmember 20.591.1.5
time.point.observer 20.591.1.2
time.point.special 20.591.1.4
time.syn 20.578.2
time.traits 20.585.1
time.traits.duration_values 20.585.1.2
time.traits.is_fp 20.585.1.1
time.traits.specializations 20.585.1.3
tuple 20.34
tuple.assign 20.51.0.1
tuple.cnstr 20.37.1.1

tuple.creation 20.53.0.2
tuple.elem 20.57.0.1
tuple.general 20.34.1
tuple.helper 20.56.0.1
tuple.rel 20.61.0.1
tuple.special 20.65.0.1
tuple.swap 20.53.0.1
tuple.traits 20.64.0.1
tuple.tuple 20.37.1
type.descriptions 17.17.1.1
type.descriptions.general 17.17.1.1.1
type.index 20.610
type.index.hash 20.611.1
type.index.members 20.610.3
type.index.overview 20.610.2
type.index.synopsis 20.610.1
type.info 18.227.1

U

uncaught 18.253.1
underflow.error 19.7.2
unexpected D.62.4
unexpected.handler D.62.1
uninitialized.copy 20.116.0.2
uninitialized.fill 20.116.0.3
uninitialized.fill.n 20.116.0.4
unique.ptr 20.127.1
unique.ptr.create 20.178.0.1
unique.ptr.dltr 20.141.0.1
unique.ptr.dltr.dflt 20.144.0.0.1
unique.ptr.dltr.dflt1 20.144.0.0.2
unique.ptr.dltr.general 20.141.0.1.1
unique.ptr.runtime 20.177.0.1
unique.ptr.runtime.ctor 20.178.0.0.1
unique.ptr.runtime.modifiers 20.178.0.0.3
unique.ptr.runtime.observers 20.178.0.0.2
unique.ptr.single 20.144.0.1
unique.ptr.single.asgn 20.165.0.0.1
unique.ptr.single.ctor 20.150.0.0.1
unique.ptr.single.dtor 20.157.0.0.1
unique.ptr.single.modifiers 20.171.0.0.1
unique.ptr.single.observers 20.170.0.0.1
unique.ptr.special 20.189.0.1
unord 23.932
unord.general 23.932.1
unord.hash 20.443.1
unord.map 23.932.4
unord.map.cnstr 23.933.0.1
unord.map.elem 23.933.0.2
unord.map.modifiers 23.936.0.1
unord.map.overview 23.932.4.1
unord.map.swap 23.936.0.2
unord.map.syn 23.932.2
unord.multimap 23.936.1
unord.multimap.cnstr 23.937.0.1
unord.multimap.modifiers 23.937.0.2
unord.multimap.overview 23.936.1.1
unord.multimap.swap 23.937.0.3
unord.multiset 23.938.1
unord.multiset.cnstr 23.938.1.2
unord.multiset.overview 23.938.1.1
unord.multiset.swap 23.938.1.3
unord.req 23.437.1
unord.req.except 23.539.0.1
unord.set 23.937.1
unord.set.cnstr 23.938.0.1
unord.set.overview 23.937.1.1
unord.set.swap 23.938.0.2
unord.set.syn 23.932.3
upper.bound 25.173.0.1
using 17.109.1
using.headers 17.110.0.1
using.linkage 17.125.0.1
using.overview 17.109.1.1
usrlit.suffix 17.424.0.0.1
util.dynamic.safety 20.85.1
util.smartptr 20.189.1
util.smartptr.enab 20.266.0.2
util.smartptr.getdeleter 20.263.0.0.1
util.smartptr.hash 20.282.0.1
util.smartptr.ownerless 20.266.0.1
util.smartptr.shared 20.189.1.2
util.smartptr.shared.assign 20.237.0.0.1
util.smartptr.shared.atomic 20.279.0.1
util.smartptr.shared.cast 20.262.0.0.3
util.smartptr.shared.cmp 20.262.0.0.1
util.smartptr.shared.const 20.201.0.0.1
util.smartptr.shared.create 20.247.0.0.1
util.smartptr.shared.dest 20.229.0.0.1
util.smartptr.shared.io 20.264.0.0.1
util.smartptr.shared.mod 20.241.0.0.1
util.smartptr.shared.obs 20.241.0.0.2
util.smartptr.shared.spec 20.262.0.0.2
util.smartptr.weak 20.264.0.1
util.smartptr.weak.assign 20.264.0.1.3
util.smartptr.weak.const 20.264.0.1.1
util.smartptr.weak.dest 20.264.0.1.2
util.smartptr.weak.mod 20.264.0.1.4
util.smartptr.weak.obs 20.264.0.1.5
util.smartptr.weak.spec 20.266.0.0.1
util.smartptr.weakptr 20.189.1.1

utilities 20
utilities.general 20.1
utility 20.2
utility.arg.requirements 17.128.1.1
utility.exchange 20.4.1
utility.requirements 17.128.1
utility.swap 20.3.2

V

valarray.access 26.139.0.1
valarray.assign 26.137.0.1
valarray.binary 26.141.1.1
valarray.cassign 26.141.0.3
valarray.comparison 26.141.1.2
valarray.cons 26.136.0.1
valarray.members 26.141.0.4
valarray.nonmembers 26.141.1
valarray.range 26.142.7
valarray.special 26.142.0.1
valarray.sub 26.141.0.1
valarray.syn 26.129.1
valarray.transcend 26.141.1.3
valarray.unary 26.141.0.2
value.error.codes 17.685.0.2
vector 23.696.1
vector.bool 23.890.1
vector.capacity 23.758.0.1
vector.cons 23.748.0.1
vector.data 23.801.0.1
vector.modifiers 23.806.0.1
vector.overview 23.696.1.1
vector.special 23.890.0.1

W

wide.stream.objects 27.53.1

X

xref F

Y

Z